VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Detekce návrhových vzorů ve zdrojových kódech .NET

Design Pattern Detection in .NET Programming Code

2011

Václav Bazgier

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 4. května 2011 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2011 .....

Děkuji Mgr. Miloši Kudělkovi PhD. za vedení mé diplomové práce, poskytnutou volnost při realizaci a za velice cenné rady při konzultacích.

Abstrakt

Cílem této diplomové práce bylo nahlédnout do problematiky detekce návrhových vzorů ve zdrojových kódech platformy .NET. Seznámení se s problematikou převodu komponent a vzorů do matice a provedení analýzy pomocí vybraných metod.

Klíčová slova: vzory, návrhové vzory, matice, detekce, reflexe, .NET framework,Similarity scoring, detekce vzorů

Abstract

The main objective of this diploma thesis is to introduce to design pattern detection. In this work is short description of design pattern and detection technics. Then is implemente one of algorithm Similarity scoring on .NET framework.

Keywords: pattern, design patterns, matrix, reflection,.NET framework, Similarity scoring,patterns detection

Obsah

1 Úvod **7**

2 Návrhové vzory **8**
2.1 Historie vzorů . . . 8
2.2 Jak vzory vznikají . . . 8
2.3 Rozdělení návrhových vzorů . . . 9
2.3.1 Vytvářející vzory . . . 9
2.3.1.1 Abstraktní továrna . . . 10
2.3.1.2 Stavitel . . . 10
2.3.1.3 Tovární metoda . . . 11
2.3.1.4 Prototyp . . . 11
2.3.1.5 Singleton . . . 12
2.3.2 Strukturální vzory . . . 12
2.3.2.1 Adaptér . . . 13
2.3.2.2 Dekorátor . . . 13
2.3.2.3 Kompozit . . . 14
2.3.2.4 Fasáda . . . 15
2.3.2.5 Proxy . . . 16
2.3.3 Vzory chování . . . 17
2.3.3.1 Příkaz . . . 17
2.3.3.2 Iterátor . . . 18
2.3.3.3 Pozorovatel . . . 18
2.3.3.4 Prostředník . . . 19
2.3.3.5 Strategie . . . 20
2.4 Další rozdělení vzorů . . . 21
2.4.1 Application architecture patterns . . . 21
2.4.2 Data access patterns . . . 21
2.4.3 Enterprise Integration Patterns . . . 22
2.4.4 User Interface Patterns . . . 22
2.4.5 Security patterns . . . 22

3 Teorie grafů **23**
3.1 Reprezentace grafu v počítači . . . 24

4 Matice **25**
4.1 Reprezentace grafu jako matice . . . 25
4.2 Převod vzoru do grafu . . . 26

5 Vzory vhodné pro analýzu **28**
5.1 Vzory z kategorie Application architecture patterns . . . 28

6 .NET framework **30**
6.1 Historie . . . 30
6.2 Common Intermediate Language . . . 31
6.3 Reflexe . . . 31
6.4 Převod zdrojového kódu do grafu . . . 32

7 Metody detekce **33**
7.1 Abstraktní syntaktický strom . . . 33
7.2 Abstraktní sémantický graf . . . 33
7.3 Struktura vzoru . . . 34
7.4 Chování vzoru . . . 34
7.5 Sémantika vzoru . . . 35
7.6 Přehled metod detekce . . . 35
7.6.1 Fingerprinting design pattern . . . 35
7.6.2 Design pattern mining enhanced by machine learning . . . 35
7.6.3 Efficient Identification of Design Patterns with Bit-vector Algorithm 35
7.6.4 A static reference analysis to understand design pattern . . . 36
7.6.5 Design Pattern Detection By Template Matching . . . 36

8 Metody detekce pomoci grafů **37**
8.1 Složitost . . . 37
8.2 Přesné metody . . . 37
8.3 Přibližné metody . . . 39
8.3.1 Editační vzdálenost . . . 39
8.3.2 Neuronové sítě . . . 39
8.3.3 Fuzzy množiny . . . 39
8.3.4 Similarity scoring . . . 40

9 Uživatelská příručka **44**

10 Popis implementace **49**
10.1 Specifikace požadavků . . . 49
10.2 Návrh aplikace . . . 49
10.2.1 Diagram tříd . . . 49
10.2.2 Diagram vrstev . . . 50
10.3 Klíčové funkce . . . 51
10.3.1 Načtení assembly . . . 51
10.3.2 Sestavení matic asociace a generalizace . . . 51
10.3.3 Detekce vzoru na základě similarity scoring . . . 51
10.4 Slovní popis implementace . . . 52
10.5 Popisy metod . . . 53
10.5.1 FileOpertion . . . 53
10.5.2 LoadAssembly . . . 53
10.5.3 Metody třídy MathOperation . . . 54

10.5.4 Metody třídy MatrixMake 54
10.5.5 Metody třídy SimilarityScoring 55
10.5.6 Metody třídy PatternAnalyze 55
10.5.7 Metody třídy ExportToCSV 55

11 Experimenty **56**
11.1 Pozitivní příklady 56
11.2 Negativní případy 56
11.3 Vlastní experimenty 56
11.3.1 Experiment č. 1 56
11.3.2 Experiment č. 2 57
11.3.3 Experiment č. 3 57
11.3.4 Experiment č. 4 58
11.3.5 Experiment č. 5 59
11.4 Shrnutí experimentů 59

12 Závěr **61**

13 Conclusions **62**

14 Literatura **63**

Přílohy **66**

A Dodatek **66**

Seznam tabulek

1 Pořadí tříd vzoru Dekorátor 26
2 Vybrané metody detekce 35
3 Experiment č. 1 57
4 Experiment č. 2 57
5 Experiment č. 3 58
6 Experiment č. 4 58
7 Experiment č. 5 59

Seznam obrázků

1 Návrhový vzor Abstraktní továrna . . . 10
2 Návrhový vzor stavitel . . . 10
3 Návrhový vzor Tovární metoda . . . 11
4 Návrhový vzor Prototyp . . . 12
5 Návrhový vzor Singleton . . . 12
6 Návrhový vzor Adaptér . . . 13
7 Návrhový vzor Dekorátor . . . 14
8 Návrhový vzor Kompozit . . . 15
9 Návrhový vzor Fasáda . . . 16
10 Návrhový vzor Proxy . . . 16
11 Návrhový vzor Příkaz . . . 18
12 Návrhový vzor Iterátor . . . 18
13 Návrhový vzor Pozorovatel . . . 19
14 Návrhový vzor Prostředník . . . 20
15 Návrhový vzor Strategie . . . 20
16 Graf vzoru dekorátor - asociace . . . 26
17 Graf vzoru dekorátor - generalizace . . . 27
18 Návrhový vzor Transaction script . . . 29
19 Návrhový vzor Query Object . . . 29
20 Návrhový vzor Front Controller . . . 29
21 Část systému a převedený AST . . . 34
22 Výsledný ASG . . . 34
23 Grafové rozdělení metod detekce . . . 38
24 Část systému a jednoduchý vzor . . . 38
25 Úvodní obrazovka aplikace . . . 45
26 Aplikace po načtení komponenty . . . 45
27 Aplikace s detailem vybraného vzoru . . . 46
28 Výsledky analýzy . . . 46
29 Výsledky analýzy - zobrazení vzoru . . . 47
30 Výsledky analýzy - zobrazení matice . . . 47
31 Analýza pomocí agregátoru . . . 48
32 Export do formátu CSV . . . 48
33 Datový tok . . . 50
34 Diagram tříd . . . 50
35 Diagram vrstev . . . 50
36 Načtení assembly . . . 51
37 Sestavení matic asociace a generalizace . . . 52
38 Detekce vzoru na základě similarity scoring . . . 52

Seznam výpisů zdrojového kódu

1 Algoritmus Similarity Scoring 42

1 Úvod

Během návrhu a následného vývoje software se velice často stává, že problémy, které se vyskytují, mají stejné řešení. V roce 1991 se této problematice věnovala skupina programátorů, kteří této problematice dali jistý řád a pravidla. O historii pojednává další kapitola. Návrhové vzory (dále jen vzory) hrají v dnešní době velmi důležitou roli při návrhu a následném vývoji softwaru především v komerční sféře. Znalost a použití vzorů napomáhá softwarovým architektům a analytikům řešit každodenní problémy při návrhu aplikací. Vzory popisují postupy, které se během vývoje často opakují, a které se velice často používají pro řešení konkrétního problému. Nejedná se však o nějakou část zdrojového kódu nebo komponentu, která se připojí k programu, ale jedná se o textový (většinou doplněný UML diagramem) popis řešení, jak daný problém vyřešit. Použití vzorů při vývoji software je také důležitým měřítkem při hodnocení kvalit a technologické zralosti softwarové společnosti. V rámci modelu CMMI (Capability maturity model integrity) je použití vzorů vhodné od stupně tři a výš. Model CMMI popisuje kvalitu organizace v rámci vývoje software. Model byl poprvé zaveden v roce 1990 a má pět úrovní. Každá úroveň modelu popisuje zralost vývojového týmu a tím vlastně poskytuje měřítko kvality dané organizace. Úrovně modelu jsou tyto:

1. Počáteční (Initial) - tým nerealizuje žádný proces vývoje nebo jen velice minimálně
2. Řízené (Managed) - vývojový proces je řízený a činnosti s tím spjaté také
3. Definované (Defined) - postupy jsou definovány a podrobně dokumentovány
4. Kvantitativní řízení (Quantitaviely managed) - procesy jsou řízeny a kontrolovány kvantitativně
5. Optimalizující (Optimizing) - vývojový tým neustále optimalizuje svou činnost

Model CMMI je nepostrádatelnou součástí v každé společnosti, která se snaží vyvíjet kvalitní a moderní software.

V dnešní době je vhodné posoudit a zpětně analyzovat vytvořené aplikace, zdali během vývoje softwarového systému byly použity návrhové vzory, které napomáhají analytikům nebo vývojářům řešit problémy při vývoji systému. Analýza je možná buď „okem“ zkušeného softwarového experta anebo pomocí nějakých algoritmů. Proto je cílem této diplomové práce seznámení se s postupy automatizované detekce vzorů ve zdrojových kódech na platformě .NET. Práce se stručně zaměřuje na popis jednotlivých metod detekce, a dále se podrobněji zabývá jednou zvolenou metodou. Tato metoda se nazývá Similarity Scoring, kterou jsem si vybral na základě článku v žurnálu IEEE transactions on software engineering z roku 2006. Tato metoda využívá teorii grafů, proto jsou zde grafy definovány a je zde také definována jejich reprezentace pomoci matice. V druhé části textu je pak uvedený podrobnější popis aplikace, která v rámci této práce vznikla. Na závěr je pak uvedena sada experimentů, která byla provedena na různých vstupních datech. Data byla buď speciálně pro tuto práci vytvořená nebo byly otestovány reálné aplikace (některá data jsou na přiloženém CD).

2 Návrhové vzory

Jak již bylo zmíněno v úvodní kapitole, návrhové vzory jsou velice důležité při návrhu a následné implementaci softwarové aplikace. Vzory můžeme dělit do různých kategorií, které si v následujících podkapitolách uvedeme. Nejačastěji se vzory prezentují jako schéma tříd a vezb mezi němi, to ale často nestačí a proto se ještě uvádí slovní popis. Mezi základní informace o vzoru bychom tak mohli zařadit:

- **Název** - jednoznačné vzor určuje a co nejpřesněji vystihuje jeho podstatu
- **Problém** - úvod do problému, který vzor řeší
- **Podmínky** - všechny možné situace, které mohou vzor ovlivnit. Některé situace mohou použití vzoru pomoci a jiné naopak použití vzoru zhoršují
- **Řešení** - popis jak daný vzor řeši konkrétní situaci. Obsahuje detailní popis všech nutných kroků
- **Příklady** - ukázková řešení různých situací, ve kterých se daný vzor může použít. Jedná se tak třeba i o konkrétni ukázku implementace ve zvoleném programovacím jazyce
- **Struktura** - grafické znázornění vzoru
- **Souvísející vzory** - obsahuje seznam dalších vzorů, kterých se týka řešení dané problematiky

2.1 Historie vzorů

Historie vzorů je spojena se stavební architekturou. V této souvislosti se totiž poprvé použil termín „návrhový vzor“. Vyslovil ho architekt Christopher Alexander v šedesátých letech dvacátého století. Jednalo se o popis architektonický stylů, které měly vlastně stejný původ, ale rozdílnou vlastní realizaci. Třeba gotika, baroko a další.

Ve světě počítačů se s tímto termínem poprvé setkáváme v roce 1987 na konferenci OOPSLA v Orlandu (ve státě Florida). Použili jej pánové Kent Beck a Ward Cunningham na základě několika připravených vzorů v jazyce SmallTalk, které sloužily pro začínající programátory v tomto jazyce.

Dalším významným milníkem pro zavedení vzorů bylo utvoření skupiny Gang Of Four na začátku devadesátých let dvacátého století, konkrétně v roce 1991. Skupinu GoF (jak se často zkracuje) jak samotný název napovídá, tvořilo uskupení čtyř mužů a to Ericha Gammy, Richarda Helma, Ralpha Johnsona a Johna Vlissidese, kteří se zabývali problematikou objektově orientovaného návrhu softwaru.

2.2 Jak vzory vznikají

Téměř všechna většina vzorů vzniká tak, že je vytvářejí expreti, kteří se o tuto problematiku zajímají. Jak již bylo napsáno výše, jedná se o řešení určitého problému, který se často opakuje. Fáze vytvoření návrhového vzoru by se dala shrnout do třech bodu:

- Nápad vzoru - expert sepíše a uveřejní řešení nějakého problému v rámci vývoje aplikace
- Zveřejnění - návrhový vzor se dostává do povědomí odborné veřejnosti a následně je o něm diskutováno
- Zpětná vazba - kdokoliv kdo má zájem může vzor posoudit a navrhnout úpravy nebo změny

K problematice návrhových vzorů se mohou vyjadřovat jak jednotlivci tak různé společnosti, které často mají interní postupy řešení problémů a následně je zveřejňují. Takovou společností je například Microsoft.

2.3 Rozdělení návrhových vzorů

Vzory dělíme (podle GoF) do třech hlavních kategorií:

1. Vytvářející vzory (creational patterns)
2. Strukturální vzory (structual patterns)
3. Vzory chování (behavioral patterns)

Dále jsou na ukázku uvedeny typické příklady vzorů, po jednom z každé kategorie. Pro detailnější studium vzorů lze doporučit [2] nebo [19].

2.3.1 Vytvářející vzory

Tento typ vzorů nabízí řešení, jakým způsobem vytvářet v rámci systému jednotlivé objekty. Řeší problematiku jakým způsobem vybrat správnou třídu a jak ji inicializovat a tím ji zařadit do systému.

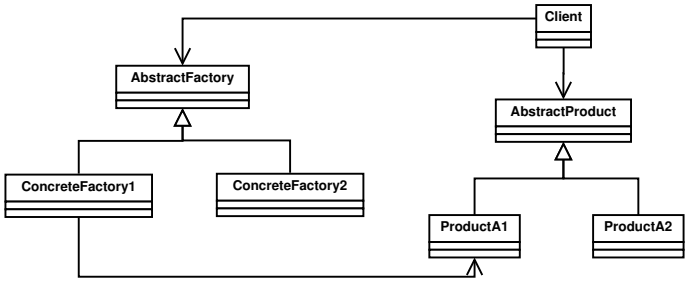
Do této kategorie patří tyto vzory:

- Abstraktní továrna (Abstract factory)
- Tovární metoda (Factory)
- Stavitel (Builder)
- Prototyp (Prototyp)
- Singleton (Singleton)

Příklady vzorů z této kapitoly jsou vzory Tovární metoda, Prototyp, Singleton a Abstraktní továrna, které jsou velice často využívány.

2.3.1.1 Abstraktní továrna

Účelem návrhového vzoru abstraktní továrna je poskytnou rozhraní pro vytváření rodin příbuzných objektů bez nutnosti specifikovat jejich třídy. Tento vzor je vyšším zobecněním návrhového vzoru Tovární metoda. Abstraktní továrna (viz. obrázek 1) je tvořena třídou `AbstractFactory`, která deklaruje rozhraní pro vytváření produktů s rozhraním deklarovaným ve třídě `AbstractProduct`. Třída `ConcreteFactory` pak implementuje odpovídající tovární metody pro vytváření konkrétních produktů požadovaných klientem.

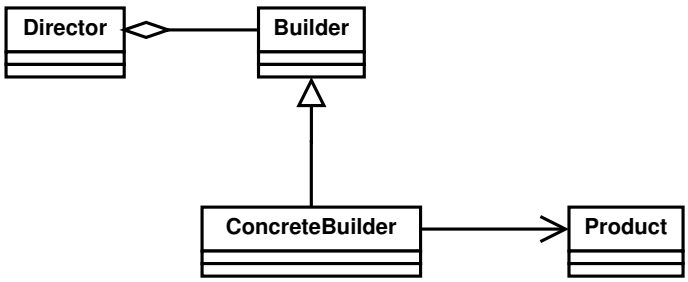


Obrázek 1: Návrhový vzor Abstraktní továrna

Se vzorem `Abstraktní továrna` často souvisí vzor `Tovární metoda` a `Singleton`.

2.3.1.2 Stavitel

Pomocí tohoto vzoru lze řešit problematiku konstrukce vzájemě podobných objektů s tím, že jejich následná prezentace je odlišná.



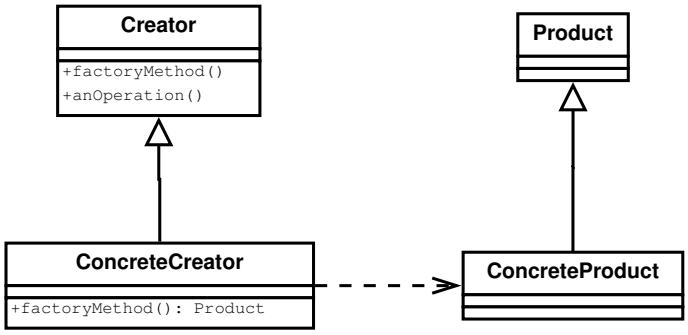
Obrázek 2: Návrhový vzor stavitel

Třída `Builder` představuje abstraktní rozhraní (viz. obrázek 2) pro vytváření částí objektu `Product`. Třídy, které implementují rozhraní `Builder` (nebo dědí z abstraktní třídy), znají proces, jak vytvořit konkrétní části produktu a dát je dohromady. Instance třídy `Director` získává od klienta určení, který `Builder` má použít. Tím je i definováno, který produkt bude vyráběn. `Director` řídí vytvoření produktu voláním metod

pro zhotovení jednotlivých částí z rozhraní `Builder`. Po vytvoření požadované části výsledného produktu je tato do něho začleněna. `Director` je odstíněn od způsobu, jakým se vytvářejí konkrétní části a jak se skládají. Určuje ovšem kdy se mají části vyrobit a tím řídí proces vytváření produktů.
S tímto vzorem je velice často spojovaný vzor `Kompozit` a dále jsou to vzory `Tovární metoda` a `Iterpreter`.

2.3.1.3 Tovární metoda

Tento návrhový vzor (někdy se také uvádí jen `Továrna`) (viz. obrázek 3) popisuje způsob vytvoření objektu, přičemž to, která instance třídy se vytvoří, rozhodne odpovídající podtřída.



Obrázek 3: Návrhový vzor Tovární metoda

Předpokladem použití toho vzoru je existence několika tříd, které mají společného předka a tyto třídy poskytují různé služby nad různými daty. Vzor továrna, pak umožňuje vybrat v průběhu běhu aplikace potřebnou instanci konkrétni třídy.

Na UML diagramu (viz. obrázek 3) lze vidět třídu `Creator`, která zastupuje vstup k tomu vzoru. Klient si vyžádá instanci třídy `ConcreteCreator`, o které třída `Creator` rozhodne v rámci vlastní logiky a požadovaných parametrů. Třída `ConcreteCreator` pak vrátí klientovi vlastní třídu typu `ConcreteProduct`.

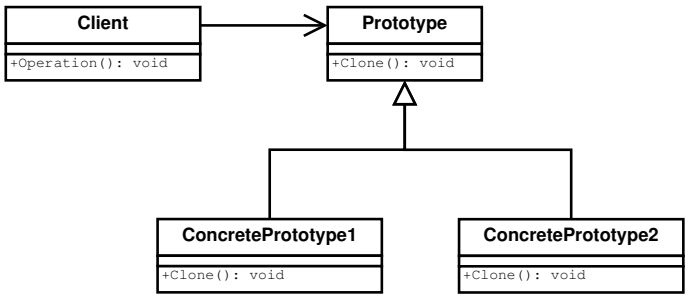
Tento vzor se používá v případech, když potřebujeme odstínit logiku vytváření instanci tříd od vlastní logiky aplikace. Na tento vzor existuje několik variací, třeba s více tříd typu `ConcreteCreator`. Typickým příkladem použití tohoto vzoru bychom mohli najít v problematice připojování k databázovým systémům.

Se vzorem `Tovární metoda` mají souvislost vzory `Abstaktní továrna` a `Šablona`.

2.3.1.4 Prototyp

Dalším typickým vzorem z kategorie vytvářejících vzorů, je vzor Prototyp. Tento vzor řeší problematiku vytváření kopie existujícího objektu aniž by musela být vytvořena nová instance třídy. Tento přístup je vhodný ve chvíli, kdy instance objektů je časově náročná,

proto je výhodné vytvořit instanci třídy a zbytek naklonovat.
Tento vzor se dá použít za předpokladu, že programovací jazyk umožňuje vytváření



Obrázek 4: Návrhový vzor Prototyp

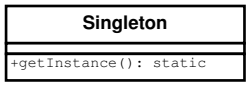
kopie již existujícího objektů.

Klient (viz obrázek 4) k tomuto vzoru přistupuje tak, že má referenci na třídu `Prototype`, ta má v sobě implementovanou metodu `Clone()`, která je schopná vytvořit vlastní kopii.

Tomuto řešení odpovídá také návrhový vzor `Továrna`, který může být použit jako alternativa. S tímto vzorem mají souvislost například vzory `Kompozit`, `Abstraktní továrna`, `Dekorátor` a `Fasáda`.

2.3.1.5 Singleton

Cílem vzoru je zajištění existence pouze jedné instance dané třídy a poskytnutí globálního přístupu k ní. Schéma tohoto vzoru tedy obsahuje pouze jednu třídu (viz. obrázek 5). Použití tohoto vzoru je odlišné v různých programovacích jazycích.



Obrázek 5: Návrhový vzor Singleton

Tento vzor se během vývoje softaware velice často používá. S tímto vzorem se také často spojují vzory `Stav`, `Fasáda`, `Prototyp`, `Abstraktní továrna` a `Stavitel`.

2.3.2 Strukturální vzory

Strukturální vzory popisují možnosti, jakým způsobem v systému uspořádat jednotlivé třídy popř. komponenty. Snaží se maximálně zpřehlednit čitelnost takto navrženého systému.

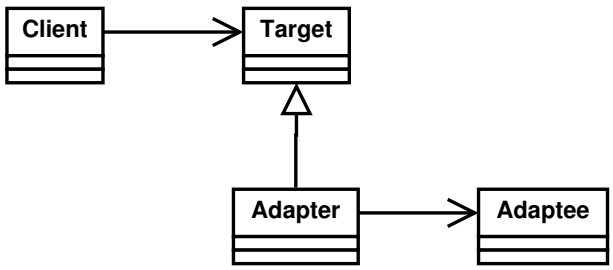
Mezi vzory do této kategorie patří:

- Adaptér (Adapter)

- Most (Bridge)
- Kompozit,strom (Composite)
- Dekorátor (Decorator)
- Fasáda (Facade)
- Muší váha (Flyweight)
- Proxy

2.3.2.1 Adaptér

S tímto vzorem lze řešit problematiku konverze rozhraní třídy ke komunikaci s jinou třídou.
Součásti návrhového vzoru Adaptér (viz. obrázek 6) je třída `Target` definující rozhraní specifické pro využití třídou `Client`. Třída `Adaptee` definuje funkcionalitu a rozhraní, které je třeba přizpusobit. Třída `Adapter` pak přizpůsobuje rozhraní třídy `Adaptee` na cílové rozhraní definované třídou `Target`.



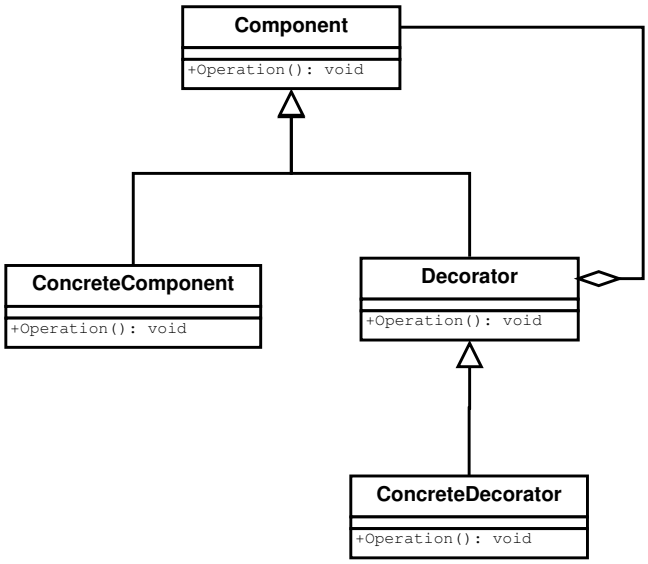
Obrázek 6: Návrhový vzor Adaptér

Vzory, které se nejčastěji používají se vzorem `Adaptér` jsou `Fasáda`, `Proxy`, `Iterátor` a `Strategie`.

2.3.2.2 Dekorátor

S pomocí tohoto vzoru je možné řešit problém změny vlastností instance třídy, bez toho aniž by se vytvořit nová odvozená třída. Dekorátor nabízí možnost jak dynamicky ovlivňovat chování objektů za běhu aplikace.

Na UML diagramu (viz. obrázek 7) je vidět třída `Component`, která reprezentuje abstraktní rozhraní pro skupinu objektů, kterým se bude dynamicky upravovat chování. Třída `ConcreteComponent` je pak konkrétni třída, se kterou se bude dál pracovat. Ve třídě `Decorator` je pak obsažena reference na `Component` a tak tuto třídu zapouzdřuje,



Obrázek 7: Návrhový vzor Dekorátor

tím pádem všechny operace, které se volají na třídě `Decorator` jsou přesněřovány na třídy `ConcreteComponent`.

Typickým použitím toho vzoru je situace, kdy k některým objektům ze stejné skupiny je zapotřebí přidat nebo odebrat nějakou další funkčnost. Použití vzoru také rozšiřuje flexibilitu funkčnosti objektů, než je při použití dědičnosti.

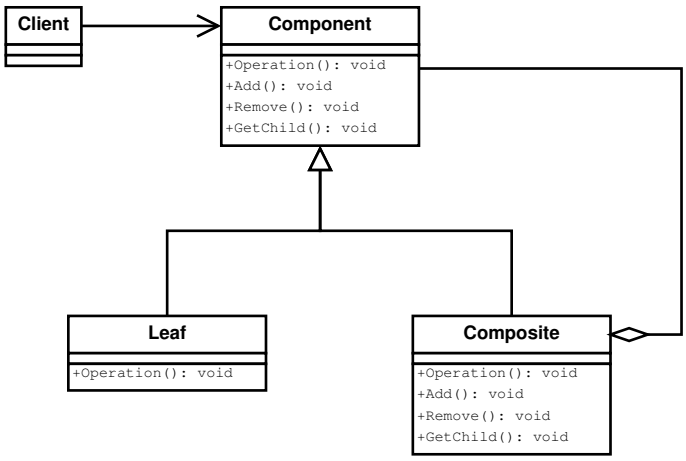
S tímto vzorem souvisí dále vzory `Kompozit`,`Strategie` a `Šablona`.

2.3.2.3 Kompozit

Někdy se také tomu vzoru říká Strom a to z toho důvodu,že řeší problematiku uspořádání objektů do stromové struktury. Řeší také problematiku přístupu k jednotlivým objektům anebo k nim složeným (kompozitní) objektům.

Složeným objektem v tomto vzoru rozumíme objekt, který je složen z jiných objektů. Tyto objekty můžou být opět složené anebo jednoduché. Jednoduchým objektem pak rozumíme takový objekt, který nemá referenci na jiný objekt.

Z UML diagramu (viz. obrázek 8) je patrné, že klient přistupuje ke třídě `Component`, která reprezentuje jednoduché rozhraní ke stromové struktuře. V tomto objektu jsou definovány metody `Add(Component), Remove(Component) atd.`,které mají za úkol pracovat s jednotlivými objekty zařazenými do struktury. Dále je ve vzoru definována třída `Composite`, která zajišťuje složený objekt. Poslední třídou ze vzoru je `Leaf`, což reprezentuje jednoduchou třídu.



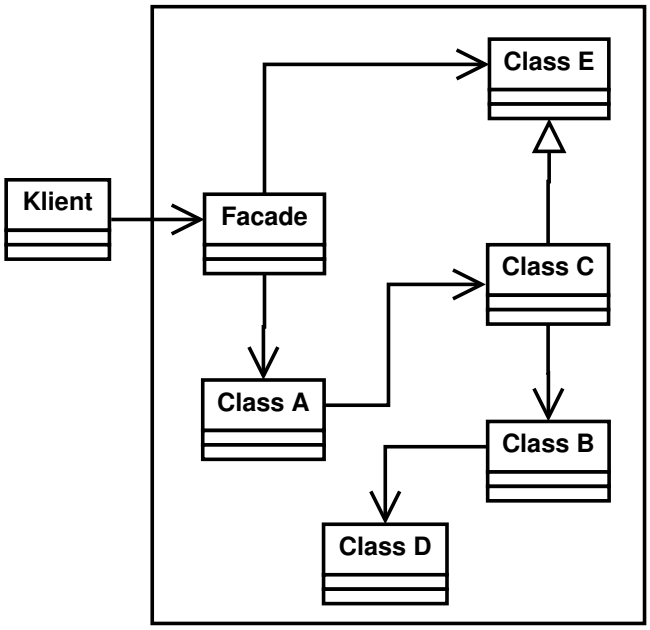
Obrázek 8: Návrhový vzor Kompozit

Využití tohoto vzoru je velice rozmanité, hodí se všude tam, kde je zapotřebí skládat objekty do stromové struktury a dále kategorizovat. Obvykle se tento vzor využívá při práci s uživatelským rozhraním.

Vzor `Kompozit` může být dále kobinován se vzory `Řetězec zodpovědnosti`, `Iterátor` anebo `Návštěvník`.

2.3.2.4 Fasáda

Tento vzor se používá v situaci, kdy je zapotřebí vytvořit jeden bod, pro přístup do systému. Tato třída má za úkol komunikovat mezi jednotlivými částmi systému, nepatří tak do vlastního systému, ale tvoří tak vyšší stupeň přístupu. Třída, která reprezentuje vzor Fasáda by neměla obsahovat žádnou funkcionalitu systému, který zastupuje (viz. obrázek 9).

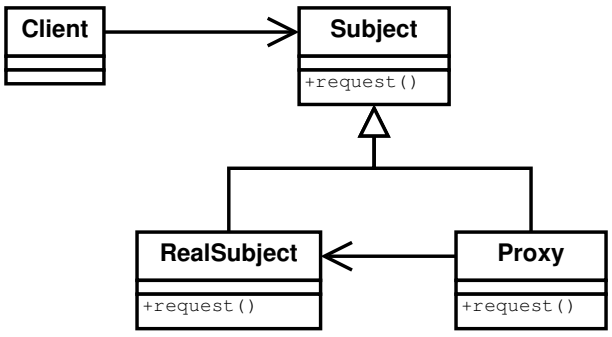


Obrázek 9: Návrhový vzor Fasáda

Užitečnost tohoto vzoru je v tom, že klient (např. jiná část systému) nemusí komunikovat s jednotlivými třídami systému, ale má tak zajištěný centráli bod. Ve spojení se vzorem `Fasáda` se často zmiňují vzory `Adaptér`,`Abstrkatní továrna` a `Singleton`.

2.3.2.5 Proxy

Úkolem návrhového vzoru Proxy je poskytnout zástupce (náhradníka) za jiný objekt a umožnit tak řízený přístupu k tomuto objektu.



Obrázek 10: Návrhový vzor Proxy

Součástí tohoto návrhového vzoru (viz. obrázek 10) je třída `Proxy` umožnující přístup instance třídy `Client` k objektu `RealSubject` výhradně přes své rozhraní. Třída `Subject` definuje společné rozhraní pro oba z uvedených objektů, tedy `RealSubject` i jeho zástupce `Proxy`. S tímto vzorem souvisí vzory `Dekorátor` a `Fasáda`.

2.3.3 Vzory chování

Vzory chování se zajímají o to, jakým způsobem se systém chová a to především, jak spolu jednotlivé algoritmy a objekty spolupracuji.

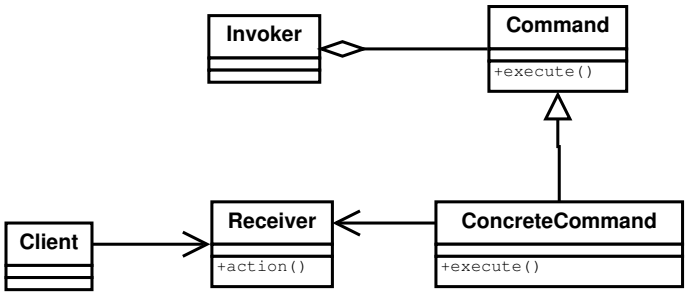
Tato kategorie vzorů obsahuje:

- Řetězec zodpovědnosti (Chain of responsibility)
- Příkaz (Command)
- Interpret (Interpreter)
- Iterátor (Iterator)
- Prostředník (Mediator)
- Memento
- Pozorovatel (Observer)
- Stav (State)
- Strategie (Strategy)
- Šablona (Template method)
- Návštěvník (Visitor)

2.3.3.1 Příkaz

Tento vzor řeší problém odstínění klienta od zpracování jeho požadavku. Klient pouze definuje požadavek a určí zpracovatele, ale už se nezajímá o způsob a čas vykonání jeho požadavku.

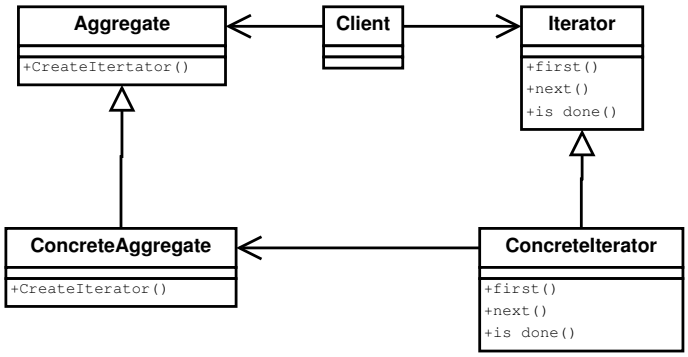
Tento návrhový vzor (viz. obrázek 11) je specifikován třídou `Command` deklarující rozhraní pro vykonání příkazu. `ConcreteCommand` je podtřídou, která definuje vazbu na příjemce příkazu (třída `Receiver`) a implementuje operaci execute cestou volání odpovídajících operací asociovaného příjemce. `Receiver` tak implementuje tyto operace, zatímco `Invoker` dává pokyn k provedení příkazu. Se vzorem Příkaz jsou často využívány vzory `Šablonová metoda`, `Memento` a `Řetezec zodpovědnosti`.



Obrázek 11: Návrhový vzor Příkaz

2.3.3.2 Iterátor

Řeší problém, jak se pohybovat mezi prvky, které jsou sekvenčně uspořádány, bez znalosti implementace jednotlivých prvků posloupnosti.



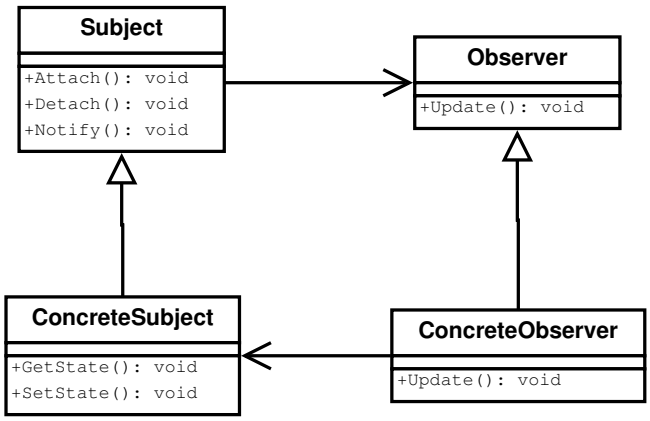
Obrázek 12: Návrhový vzor Iterátor

Návrhový vzor `Iterátor` specifikuje třídu (viz. obrázek 12) `Iterator` deklarující rozhraní pro přístup k prvkům kolekcí sekvenčním způsobem. `ConcreteIterator` implementuje převzaté rozhraní pro konkrétní typ kolekce se kterou je asociován. Třída `Aggregate` je supertřída všech typu konkrétních kolekcí (`ConcreteAggregate`).`Aggregate` deklaruje operaci `createIterator`, která je pak předefinována pro konkrétní kolekce tak, že k nim vytváří jim odpovídající iterátory. Na tento vzor často navazují vzory `Memento` a `Kompozit`.

2.3.3.3 Pozorovatel

Tímto vzorem se řeší problém závislosti jednoho objektu na dalších objektech. Jedná se vlastně o řešení problému, jakým způsobem šířit události z jednoho objektu na objekty

na něm závislými.



Obrázek 13: Návrhový vzor Pozorovatel

Vzor Pozorovatel se dá použít v situaci, kdy je definována závislost objektu na nějakém jiném objektu. Závislost je pak myšlena tak, že změny v nezávislém objektu jsou dále šířeny objektům závislým. Nezávislý objekt tak musí informovat ostatní objekty o změně událostí, které mohou závislé objekty ovlivnit. Podmínkou použití návrhového vzoru je pak také možnost udržení si seznamu závislých objektů.

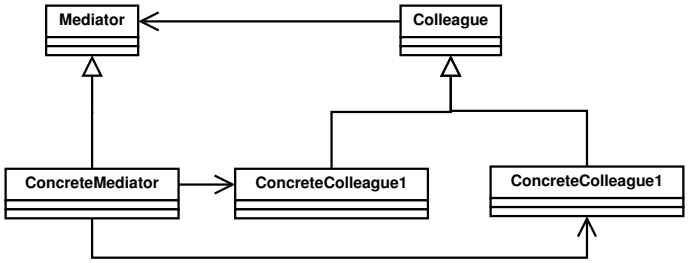
Na UML diagramu (viz. obrázek 13) je definována třída `Subject`, která reprezentuje rozhraní pro nezávislý objekt. Dále je v rámci třídy `Subject` vytvořena metoda `Notify()`, která má za úkol informovat závislé objekty o změnách. Třída `Observer` pak přesdtavuje rozhraní, které reprezentuje závislé objekty. Vlastní objekt, který je závislý na instanci třídy `Subject` je instance třídy `ConcreteObserver`. Tento objekt má pak referenci na objekt vzniklý z třídy `ConcreteSubject`.

Typickým použitím toho vzoru jsou grafická rozhraní, popřípadě různé vícevrstvé architektury, například `MVC`. S tímto vzorem jsou pak spojeny vzory jako je `Řetězec zodpovědnosti`,`Prostředník`,`Příkaz`.

2.3.3.4 Prostředník

Častěji se název tohoto vzoru udává v anglickém jazyce a to Mediator. Vzor nabízí řešení komunikace dvou komponent systému aniž by navzájem znaly své metody. Velice často se tento vzor používá v kombinaci s již prezentovaným vzorem Pozorovatel.

Použití vzoru Prostředník je jedním z řešení problematiky komunikace objektů nebo komponent mezi sebou. V rozsáhlých systémech není možné, aby jednotlivé objekty anebo komponenty komunikovaly napřímo, proto komunikace probíhá zprostředkovaně.



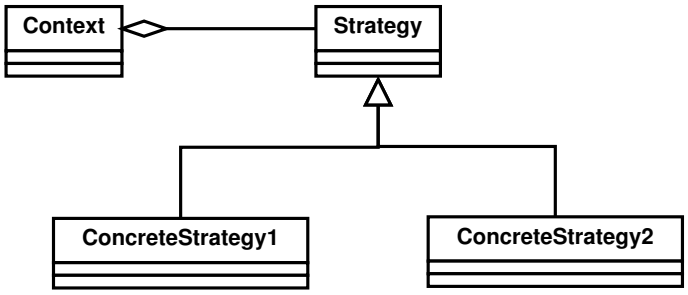
Obrázek 14: Návrhový vzor Prostředník

Na UML diagramu (viz. obrázek 14) lze vidět třídu `Mediator`, která definuje komunikaci jednotlivých tříd s třídou `ConcreteMediator`, která zapouzdřuje komunikaci mezi objekty - komponentami. Třídy `ConcreteColleague` jsou pak samotné třídy, které by mezi sebou komunikovaly, ale místo toho využívají služeb prostředníka.

Tomuto vzoru jsou velmi podobné vzory `Pozorovatel` a `Fasáda`.

2.3.3.5 Strategie

Účelem návrhového vzoru Strategie je definovat rodinu algoritmu zapouzdřených do objektu a zajistit tak jejich zaměnitelnost. Tento vzor umožnuje zaměňovat algoritmy nezávisle na objektu, který je využívá.



Obrázek 15: Návrhový vzor Strategie

Návrhový vzor `Strategie` (viz. obrázek 15) je tvořen třídou `Strategy` deklarující rozhraní pro použití konkrétních algoritmů využívaných třídou `Context`. `ConcreteStrategy` implementuje tyto algoritmy. Třída `Context` má pak jediný úkol, a to udržovat vazbu na aktuálně vybraný algoritmus. Vzor `Strategie` se obvykle využívá se vzory `Most`, `Šablonová metoda` a `Stav`.

2.4 Další rozdělení vzorů

Důležité je také podotknout, že rozdělení podle GoF, není jediné možné rozdělení. Existují další rozdělení vzorů podle různých témat a zaměření. Mezi další kategorie patří například Application architecture patterns, Data access patterns, Enterprise Integration Patterns a User Interface Patterns. V následujících podkapitolách si jentotlivé kategorie vzorů krátce představíme.

2.4.1 Application architecture patterns

Toto rozdělení vzorů vychází z knihy Patterns of Enterprise Application Architecture od autora Martina Fowlera (viz. [15]). Vzory jsou rozděleny do skupin, které v rámci aplikace zastupují různé komponenty systému. Skupiny jsou následující:

- Domain logic patterns
- Data source architecural patterns
- Object-relational behavioral patterns
- Object-relational structural patterns
- Object-relational metadata mapping patterns
- Web presenation patterns
- Distribution patterns
- Offline concurrency patterns
- Session state patterns
- Base patterns

2.4.2 Data access patterns

Tento typ vzorů se zabývá různými metodami přístupu k datům. Vzory se v tomto rozdělení nedělí dál na kategorie, proto si vyjmenujeme nejvýznamnější vzory, které do této kategorie patří:

- Cache patterns
- Resource patterns
- Input/output patterns
- Decoupling patterns
- A další

2.4.3 Enterprise Integration Patterns

Tato kategorie vzorů popisuje integraci složitých a rozsáhlých podnikových systémů. Vzory popisují technologický postup samotných integrací. Tyto vzory nejčastěji najdou uplatnění při budování bankovních a různých ekonomických systémů. Více se o těchto vzorech je možné se dozvědět například zde. Vzory jsou rozděleny do těchto kategorii:

- Integration Styles
- Messaging Systems
- Messaging Channels
- Message Construction
- Message Routing
- Message Transformation
- Messaging Endpoints
- System Management

2.4.4 User Interface Patterns

Další významnou kategorií, která rozděluje návrhové vzory je kategorie vzorů uživatelského rozhraní. Tato kategorie vzorů popisuje řešení a techniky, jakým způsobem navrhnout a realizovat uživatelské rozhraní. Zabývá se jak uživatelským rozhraním pro webové aplikace tak aplikace desktopové. Základní dělení vzorů:

- Organizing the Content
- Getting Around
- Organizing the Page
- Getting Input From Users
- Showing Complex Data
- Commands and Actions
- Direct Manipulation
- Stylistic Elements

2.4.5 Security patterns

Zajimavou skupinou návrhových vzorů jsou vzory bezpečnosti. Každá vzor z kategorie GoF má svůj bezpečnostní ekvivalent. Touto skupinou vzorů se zabývá např. [17].

Více o rozdělení vzoru například na www.pattron.net[8]. Nebo na www.objects.cz[18].

3 Teorie grafů

Při vlastní realizaci této práce se pracuje s některými důležitými pojmy z oblasti teorie grafů. V této kapitole budou popsány pouze nezbytné pojmy, pro více informací lze doporučit [3].

Teorie grafů je významnou kapitolou teoretické informatiky a také má široké praktické využití. Pomocí grafů se můžou řešit úlohy, jako je tok v síti, problém čtyř barev a jiné. Grafy se mohou reprezentovat několika způsoby:

- Diagramem
- Definicí (slovním popisem)
- Maticí
- Datovýma strukturama

Mezi nejčastěji používanou reprezentaci v počítači, bychom mohli zařadit reprezentaci pomoci matice, v důsledku tedy ve formě dvojrozměrného pole. Více si o reprezentaci napíšeme na konci této kapitoly.

Definice 3.1 ***Graf*** *je trojice $G = (H, U, P)$, kde*

- *H je množina hran $H = \{h_1, h_2, \ldots .., h_n\}$*
- *U je množina uzlů $U = \{u_1, u_2, \ldots, u_n\}$*
- *P je incidenční funkce, která je dána předpisem:*
  - *Pro neorientovaný graf: $H \to U \otimes U$, kde $U \otimes U$ označuje množinu všech neuspořádaných dvojic prvků z množiny U.*
  - *Pro orientovaný graf $H \to U \times U$, kde $U \times U$ označuje kartézský součin (množinu všech uspořádaných dvojic prvků z množiny U).*

Pro počet hran v množině H používáme zápis $|H|$ a pro počet uzlů v množině U používáme zápis $|U|$.

Graf může také obsahovat uzly, do kterých nevedou žádné hrany, v takovém případě nazýváme graf nesouvislý.

Grafy tedy můžeme podle definice rozdělit na orientované a neorientované. Neorientovaný graf obsahuje symetrické vztahy mezi objekty, kdežto orientovaný graf má vazby nesymetrické. Orientace vztahu se v takovémto grafu vyznačuje pomocí šipky.

Dále si musíme zavést pojmy podgraf a isomorfismus grafů.

Definice 3.2 *Mějme graf $G = (H, U, P)$. Pak graf $G' = (H', U', P')$ takový, že*

- *$H' \subseteq H$ množina hran grafu G' je podmnožinou množiny hran grafu G*

- *$U' \subseteq U$ množina uzlů grafu G' je podmnožinou množiny uzlů grafu U*
- *Pro každou hranu $h \in H$ platí $p'(h) = p(h)$ incidenční funkce grafu G' je zúžením incidenční funkce grafu G*

se nazývá podgraf grafu G.

Definice 3.3 *Dva grafy G a G' jsou isomorfní právě tehdy, když existuje takové zobrazení $f : V(G) \to V(G')$, že platí $\{i, j\} \in E(G) \Leftrightarrow \{f(i), f(j)\} \in E(G')$*

3.1 Reprezentace grafu v počítači

Jak již bylo napsáno na začátku této kapitoly, matice se dají v počítači reprezentovat několika způsoby. První z nich je pomocí dvou rozměrného pole. Tato reprezentace je výhodná pro svou jednoduchost použití a snadnou představivost. Problém ale může nastat ve chvíli, kdy je matice příliš velká. V této situaci může dojít k nedostatku paměti a ztrátě dat.

Druhý způsob reprezentace je možný pomocí dvou dvou polí. V prvním poli o velikosti počtu uzlů grafu jsou uloženy indexy do druhého pole, které reprezentuji místo od kterého jsou uloženy indexy na sousední uzly daného uzlu. Je zde ještě nutné zavést oddělovač, kterým jasně řekneme, kde končí výčet daného uzlu, většinou se používá hodnota -1.

Poslední často používanou reprezentací matice je dynamická datová struktura. Tou se myslí třída, která obsahuje pole sousedních uzlů.

4 Matice

Dalším důležitým pojmem, se kterým se v této práci bude pracovat je pojem matice.

Definice 4.1 *Matice M se definuje nad číselným tělesem P jako zobrazení $\{1,2,...,n\}\times\{1,2,...,m\} \to P$. Jedná se vlastně o tabulku, která obsahuje n sloupců a m řádku, každý prvek matice se dá jednoznačné určit pomocí indexu, který je dán z rozsahu $\{1,...,n\}$ popř. $\{1,...,m\}$. Pokud $n = m$, pak hovoříme o tzv. čtvercových maticích. S tímto typem matic budeme dále pracovat. S maticemi můžeme provádět různé operace, kterým se obecně říká maticový počet.*

Mezi základní operace s maticemi, které v této práci budeme používat, jsou sčítání, násobení, dělení matic, transpozice matice a shoda matic.

Definice 4.2 *Nechť A, B jsou reálné matice tvaru $n \times n$, pak normu matice $\|A\|$ definujeme těmito vlastnostmi:*

1. $\|A\| > 0; \|A\| = 0$ *jestliže A je nulová matice*
2. $\|\alpha A\| = |\alpha|\|A\|$ *pro libovolný skalár α*
3. $\|A + B\| \leq \|A\| + \|B\|$
4. $\|AB\| \leq \|A\| \cdot \|B\|$

S maticemi se dále můžou provádět operace, jako jelibovolná záměna řádku a sloupců. Více je možné naleznou třeba zde [4].

4.1 Reprezentace grafu jako matice

Graf můžeme reprezentovat několika způsoby. Obvykly a již zde prezentovaný způsob je grafický, tzn. pomocí diagramu. Další způsob je už více formální a to pomoci matice.

K tomuto účelu se zavádí pojem matice sousednosti a její definice zní takto:

Definice 4.3 *Mějme graf $G = (H, U, P)$ s množinou hran $H = \{h_1, h_2, ..., h_n\}$ a množinou uzlů $U = \{u_1, u_2, ..., u_m\}$. Matice sousednosti je čtvercová matice řádu m. Její prvky pro orientovaný graf jsou dány předpisem:*

- $S_{jk} = 1$, *jestliže je hrana z uzlu u_j do uzlu u_k*
- $S_{jk} = 0$, *jestliže mezi uzly u_j a u_k není hrana.*

4.2 Převod vzoru do grafu

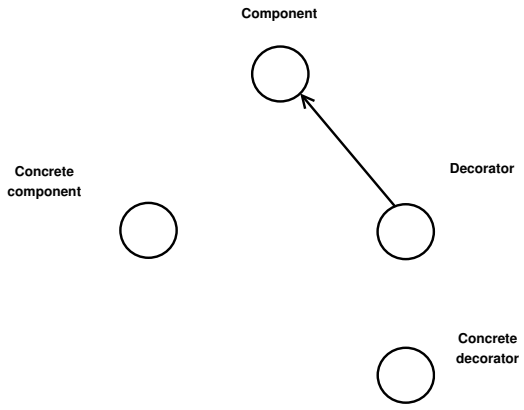
Na návrhový vzor můžeme nahlížet jako na orientovaný graf, kde jednotlivé uzly jsou třídy a orientované hrany jsou vazby mezi třídami. Při převodu vzor na graf musíme vytvořit zvlášť graf, který reprezentuje asociaci a zvlášť generalizaci mezi třídami. Důvod je ten, že operace s těmito grafy je pak jednodušší. Orientovaný graf dále převedeme na čtvercovou matici, kde řád matice je roven počtu uzlů v grafu.

Jako příklad uvedeme převod vzoru dekorátor na graf a následně na matici sousednosti. Vzor dekorátor podle obrázku (viz. obrázek 7). obsahuje čtyři třídy. Ve vzoru je zastoupena jak dědičnost, tak asociace a proto se vytvoří dva orientované grafy, které reprezentují tyto dvě vlastnosti a dále grafy převedeme na matice sousednosti.

Na začátek si označíme třídy vzoru čísly, pro jednodušší orientaci:

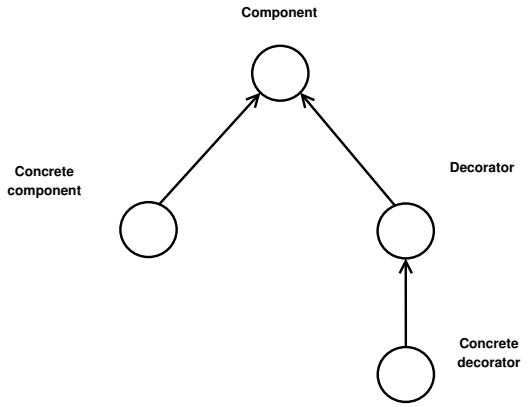
Pořadí	**Název třídy**
1.	Component
2.	ConcreteComponent
3.	Decorator
4.	ConcreteDecorator

Tabulka 1: Pořadí tříd vzoru Dekorátor



Obrázek 16: Graf vzoru dekorátor - asociace

$$A_{asociace} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$



Obrázek 17: Graf vzoru dekorátor - generalizace

$$A_{generalizace} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Z tohoto příkladu je patrný převod grafu na matice sousednosti, které nám dále umožní operace spojené s detekcí vzoru.

5 Vzory vhodné pro analýzu

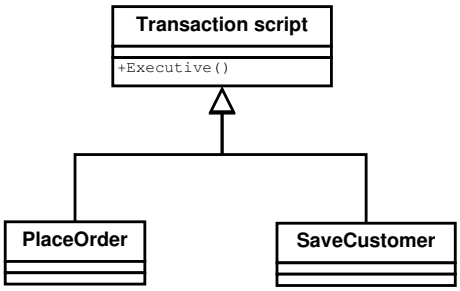
Jednotlivé vzory se dají reprezentovat jako matice za předpokladu, že ve vzorech jsou mezi třídami obsaženy vazby typu asociace a generalizace. Toto pravidlo, ale neplatí u všech vzorů, které jsou zastoupeny v rozdělení podle GoF. Vzor, který se nehodí je vzor Fasáda (Facade). Tento vzor nabízí řešení, jak zjednodušit vstup do systému. Tento vzor vytvoří vstupní bod do systému a nabízí tak rozhraní, přes které můžeme se systémem komunikovat. Vzor je ale „povýšen“ nad vlastní systém a neměl by se chápat jako součást systému z pohledu vazeb asociace a generalizace. Vzor Singleton také není vhodný pro analýzu, protože jeho úkolem je zajistit instanci pouze jednoho objektu ze zvolené třídy. Tohoto stavu se v rámci .NET frameworku dosáhne pomocí klíčového slova static a nejedná se tedy o žádnou vazbu typu generalizace/asociace.

Důležité je také zmínit velkou podobnost mezi některými vzory, která následně vede k tomu, že se v analyzované aplikaci mohou tyto vzory velice jednoduše zaměnit. Velmi podobnými vzory jsou myšleny takové vzory, které se liší třeba jen v jednom typu vazby a tím mohou být velice snad zaměnitelné. Jako příklad můžeme zmínit vzory Kompozit a Dekorátor, které se liší především v tom, že vzor dekorátor je rozšířen dědičností o třídu, která reprezentuje konkrétní třídu s rozšířenýma vlastnostmi. Další vzory které jsou si vzájemě podobné jsou vzory Prototyp a Proxy, kdy vlastně jejich jediný rozdíl je ve vazbě typu asociace u vzoru Proxy, mezi třídami, které dědí z nadtřídy.

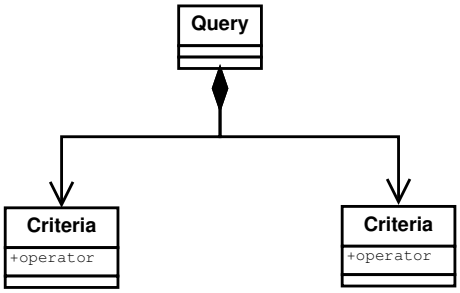
5.1 Vzory z kategorie Application architecture patterns

V rámci této diplomové práce jsem se také pokusil vyzkoušet vzory z kategorie enterprise application architecture (někdy také nazývány vzory dle Fowlera). Tyto vzory jsou v dnešní době velice aktuální, protože se často využívají při vývoji aplikací. Je tedy vhodné také je vyzkoušet detekovat pomocí algoritmu Similarity Scoring. Pokud se podíváme na strukturu těchto vzorů je velice podobná vzorům z kategorie GoF. Toto zjištění tedy dává možnost vzory převést do grafové podoby a následně sestavit matice sousednosti, které jsou vstupem pro algoritmus Similarity Scoring.

Jako příklad si uvedeme návrhový vzor **Transaction script** (viz. obrázek 18), který má za úkol organizaci logiky programu, tak aby každá procedura zpracovávala jen jeden požadavek z prezenční vrstvy.
Další návrhový vzor je **Query Object**. Tento vzor je skupina objektů, které reprezentují databázové dotazy. Na obrázku (19) je uvedeno schéma vzoru Query Object, jeho struktura je velice jednoduchá a po převodu do maticové podoby dostaneme matici o velikosti 3×3.

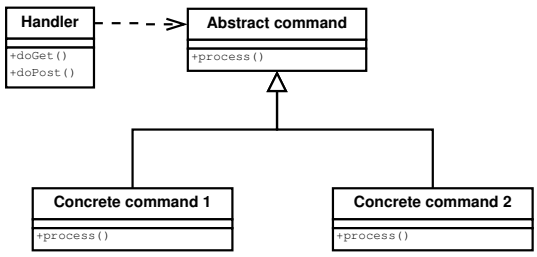


Obrázek 18: Návrhový vzor Transaction script



Obrázek 19: Návrhový vzor Query Object

Posledním vzorem z kategorie Enterprise application architecture je vzor **Front Controller**(viz. obrázek 20). Tento vzor spadá do podkategorie vzorů pro webové prezentace a jeho účelem je kontrolovat všechny žádosti o jednotlivé webové stránky. Podíváme-li se na schéma tohoto vzoru, je stejné jako schéma vzoru Prototyp. Při převodu do maticové podoby pak jsou tyto dva vzory mezi sebou nerozlišitelné. Výsledná matice pak má velikosti 4×4.



Obrázek 20: Návrhový vzor Front Controller

Tyto tři vzory, které jsou zde uvedeny, jsou zahrnuty i v aplikaci, která v rámci této diplomové práce vznikla. Problémem těchto vzorů vzhledem k jejich podobě vůči vzorům z GoF je ten, že pro algoritmus Similarity scoring není rozlišitelný zda-li se jedná o vzory z kategorie GoF anebo z kategorie Application architecture patterns.

6 .NET framework

K tomu abychom mohli analyzovat zdrojové kódy, je zapotřebí si vybrat nějaké prostředí, ve kterém se analýza realizuje. Můžeme zvolit jakýkoliv objektový jazyk, jako je třeba C++ nebo C#, ale často se stává, že komponenty a různé části systému jsou napsány v různých jazycích, což je pro potřeby analýzy nevhodné. Proto je vhodné zvolit za prostředí nějaký framework,který tyto vyšší programovací jazyky překládá do nějakého nízko úrovňového mezijazyka. Důležitou vlastností zvoleného frameworku je, aby byl framework objektový. Jedním z nejrozšířenějších frameworků pro vývoj aplikací na světě a plně objektovým prostředím je .NET framework.

Dříve než popíšeme vlastní .NET framework je nutné poznamenat, že samotný .NET framework je jen součásti platformy .NET, což je kolekce technologií vyvíjená především společností Microsoft. Platforma nabízí technologie nejen pro operační systém Windows, ale také webové technologie a také třeba technologie pro mobilní zařízení.

Samotný .NET framework je hlavní součásti platformy .NET a slouží pro provoz aplikací, tak i pro samotný vývoj těchto aplikací. Nabízí knihovny, které velice usnadňují vývoj aplikací v prostředí .NET. Hlavním vývojovým prostředkem pro platformu .NET je Visual Studio (aktuálně ve verzi 2010).

.NET framework obsahuje několik programovacích jazyků a tím nabízí vývojáři širší možnosti při vývoji samotné aplikace. Mezi nejznámější patří C#, Visual Basic a J#. Obsahuje také ale „méně obvyklé“ jazyky ve Windows, jako je třeba IronPython, F# nebo LSharp. At’ už si programátor vybere jakýkoliv jazyk, při samotném překladu programu je zdrojový kód převeden do jazyka Common intermediate language (CIL). Tomuto jazyku je věnována samostatná podkapitola dále.

6.1 Historie

.NET framework byl poprvé vydán v roce 2002 a to ve verzi 1. Součásti frameworku bylo uvedení jazyka C#, který vycházel z jazyků C++ a Java. Další novou a převratnou verzí .NET frameworku byla verze 2, která vyšla v roce 2005 a její součásti byl jazyk C# 2.0. V současné době je aktuální verze frameworku 4, které obsahuje programovací jazyk C# 3.0. Mezi důležité součásti aktuální verze frameworku můžeme řadit tyto technologie:

- ASP.NET – pro tvorbu webových aplikací
- Windows presenation foundation - technologie pro vytváření nového uživatelského
- Windows communication foundation – technologie pro tvorbu distribuovaných aplikací
- Language integrated query (LINQ) – nová objektová technologie pro přístup k datům

Samotný framework se neustále vyvíjí a nabaluje na sebe nové technologie a možnosti tvorby software. V brzké době by se měla uvolnit verze 4.5.

6.2 Common Intermediate Language

Common Intermediate Language (ve zkratce CIL) je nejnižší programovací jazyk (nazýván také někdy jako mezijazyk) v rámci platformy .NET. Do jazyka CIL se překládají všechny programovací jazyky, které jsou obsaženy v rámci .NET frameworku. Jazyk je nezávislý na hardwarové platformě. Mezi jeho základní charakteristiky a vlastnosti patří:

- Objektová orientace
- Přísná typizace dat
- Ošetření chyb prostřednictvím vyjímek
- Užití atributů

Tyto vlastnosti a přístupy daly možnost vytvořit jazyk CIL a tak poskytnout základní platformu pro .NET framework.

6.3 Reflexe

Reflexe (také se používá termín reflection) je obecný termín, který v platformě .NET obsahuje třídy a metody, pro dynamický přístup k typům a objektům v programech. Jak již bylo napsáno výše, základním stavebním kamenem .NET frameworku je matajazyk CIL, do kterého jsou zkompilovány veškeré programovací jazyky v rámci .NETu. Reflexe nám nabízí možnost sestavit zdrojový kód do metajazyka CIL, z již zkompilovaného programu. A proto můžeme zkompilovaný kód analyzovat. Protože jazyk CIL zachovává i logickou strukturu programu, můžeme tak zjistit vazby mezi jednotlivými objekty. Třídy a metody reflexe jsou dostupné ve jmenném prostoru `System.Reflection`.

Mezi důležité metody v rámci jmenného prostoru reflexe můžeme zahrnou především tyto metody a vlastnosti:

`GetTypes()`
Metoda, která zajišťuje přístup k metadatům nějakého typu.

`Type[]`
Kolekce typů, které jde jednotlivě procházet a u každého typu lze využít jeho vlastnosti. V rámci implementace byly na těchto typech nejčastěji využívany vlastnosti:

- `Name` - vrací název typu
- `isClass` - vrací ano/ne je-li typ třídou
- `isAbstract` - vrací ano/ne je-li typ abstraktní třídou
- `isInterface` - vrací ano/ne je-li typ rozhraním
- `BaseType.Name` - vlastnost v sobě nese název nadobjektu zkoumaného typu

`GetFields()`
Díky této metodě se můžeme dozvědět podrobnosti o datových členech nějakého typu.

`FieldInfo`
Tato třída umožňuje přístup k jednotlivým členům `GetFields` a umožňuje přístup k matainformacím o jednotlivým atributech. Tato třída byla využita k identifikaci vazeb asociací. Ke zjištění asociace byla využita vlastnost `FieldType.Name`.

Toto je jen stručný výklad nejdůležitějších metod a vlastností, které potřebujeme k této práci. Podrobněji se těmto metodám budeme věnovat v kapitole věnované implementaci aplikace PatFinder.

6.4 Převod zdrojového kódu do grafu

Jak již bylo napsáno výše, reflexe a metajazyk CIL nám slouží k vytvoření seznamu tříd a rozhraní a následně lze sestavit grafy reprezentující vazby mezi nimi. Pro to, abychom mohli analyzovat zdrojový kód aplikace pomocí reflexe, musí být program tedy přeložen do binární podoby. Jakmile je tedy program (také komponenta) přeložená do binární podoby, je tedy možné pomocí reflexe ji analyzovat. Reflexe nám umožňuje získat kompletní seznam tříd a dále zjistit vazby mezi jednotlivými třídami. Na základě těchto tříd a vazeb je pak možné sestavit grafy, které reprezentují analyzovaný program. Grafy se popisují pomocí matic sousednosti, se kterými se pracuje dál.

7 Metody detekce

K problematice detekce návrhového vzoru lze přistoupit několika různými způsoby, které si v následujících podkapitolách rozebereme. Než ale začneme, je vhodné si také uvést způsoby reprezentace vztahů mezi objekty a další potřebné informace v analyzovaném systému. Protože v rámci této práce vznikla aplikace, která implementuje algoritmus SimilarityScoring (viz. dále) je zde pracováno s maticí jako s reprezentací vstupních dat. Tato reprezentace se v informatice realizuje pomocí datové struktury vícerozměrné pole. Mezi další reprezentace se řadí:

- Abstraktní syntaktický strom (AST)
- Abstraktní sémantický graf (ASG)
- A další (např. Eulerův model)

V následujících odstavcích si popíšeme dvě možné reprezentace, včetně demonstrace na příkladu.

7.1 Abstraktní syntaktický strom

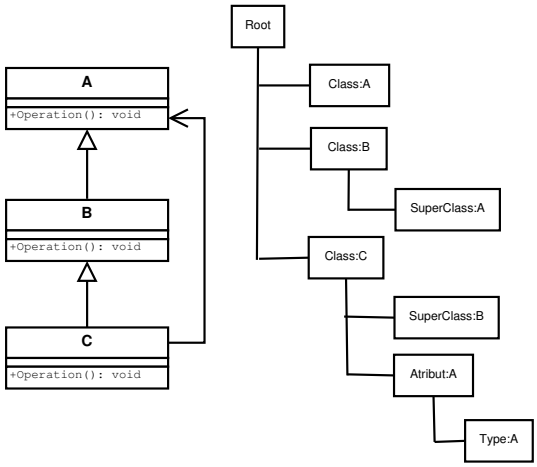
Tato datová struktura vychází z teorie překladačů, pomocí které se realizuje struktura programu během překladu do binární podoby. Jedná se ted o konečný orientovaný strom, kde každý uzel reprezentuje třídu analyzovaného systému a jeho potomci jsou uzly, které reprezentují atributy, metody a dědičnost. Na UML diagramu jednoduchého systému si ukážeme převod na AST viz. obrázek 21. Tato reprezentace se dá v programovém kódu velice dobře implementovat za použití návrhového vzoru kompozit.

7.2 Abstraktní sémantický graf

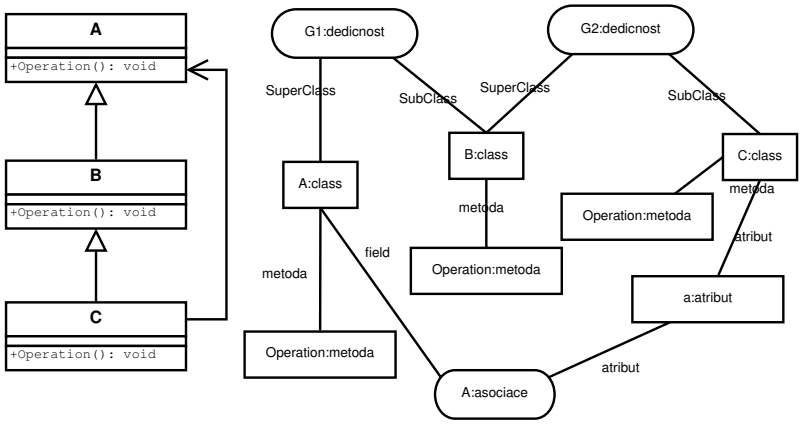
V této reprezentaci se využívají dva typy uzlů a to uzel element a uzel vztahu. Uzlem element jsou myšleny elementy analyzovaného systému jako jsou třídy, metody, atributy. Uzlem vztahu je pak myšlený vztah mezi jednotlivými elementy, tedy vztah generalizace a asociace. Na obrázku 21 a 22 je jednoduchá ukázka převodu systému na abstraktní sémantický graf.

Jak již bylo napsáno výše, problematikou detekce se zabývá více různých metod. Tyto metody vycházejí z různých pohledů na charakteristiku návrhových vzorů. Vzory lze tedy chápat a popsat takto:

- Struktura vzorů
- Chování vzoru v rámci systému
- Kombinací strukturou a chováním vzorů
- Sémantikou vzoru



Obrázek 21: Část systému a převedený AST



Obrázek 22: Výsledný ASG

7.3 Struktura vzoru

Tato charakteristika je nejčastěji využívanou (existuje několik metod), je to z důvodu toho, že tato charakteristika pouze zkoumá vztahy mezi objekty a následně tak sestaví datovou strukturu, která systém reprezentuje. Pomocí tohoto přístupu se může detekovat většina návrhových vzorů dle rozdělení GoF. Mezi ty, které lze identifikovat hůře patří vzory, které patří do skupiny vzorů chování.

7.4 Chování vzoru

Tato charakteristika na vzory nenahlíží z pohledu vazeb mezi objekty, ale z pohledu toho, jak návrhový vzor ovlivňuje chování analyzovaného systému. Tato charakteristika se téměř nikdy nepoužívá samostatně, ale v kombinaci s charakteristikou struktury vzorů.

7.5 Sémantika vzoru

Sémantika vzoru nemá přesnou definici. Tato charakteristika se snaží jemným způsobem rozlišit části systému a tak vytvořit celkový náhled na zkoumaný systém. Například se zkoumá zda dva objekty jsou spojeny k sobě vazbou $1:1$ nebo $1:n$ a zda-li jsou vztahy povinné nebo nepovinné. Tyto informace pak slouží k vytipování potenciálně použitých vzorů v aplikaci.

7.6 Přehled metod detekce

V následující tabulce je přehled vybraných metod rozdělených podle toho, jakou charakteristiku vzorů potřebují k identifikaci.

Charakteristika vzoru	**Metoda**
Struktura vzoru	Fingerprinting design pattern [20] Design pattern mining enhanced by machine learning [21] Efficient Identification of Design Patterns with Bit-vector Algorithm [22] Design Pattern Detection Using Similarity Scoring [1]
Chování vzoru	A static reference analysis to understand design pattern behavior [23]
Kombinace metod	Design Pattern Detection By Template Matching [24]

Tabulka 2: Vybrané metody detekce

7.6.1 Fingerprinting design pattern

Tato metoda využívá pro popis systému mikro-architekturu, touto architekturou je popsán i vzor (využívá struktury XML). Pak jsou tyto achitektury porovnávány a zjišťovány podobnosti. O této metodě je více napsáno v [20].

7.6.2 Design pattern mining enhanced by machine learning

Metoda je založena na stanovení možných kandidátů, které jsou vhodným způsobem popsány. Popsány systém je následně pomocí strojového učení vyhodnocen. První přístup se zabývá využitím neuronové sítě, a to konkrétně algoritmu založeném na backpropagation. Druhý přístup využívá rozhodovací stromy. Podrobnosti viz. [21].

7.6.3 Efficient Identification of Design Patterns with Bit-vector Algorithm

Tento výpočet je založený na bit-vektorovém algoritmu. Vstupní strukturou do algoritmu je vektor, který jednoznačně popisuje vztahy každé třídy s ostatními třídami v rámci analyzovaného systému. Podrobnosti viz. [22].

7.6.4 A static reference analysis to understand design pattern

Metoda, která převádí analyzovaný systém a jednotlivé metody na graf, který dálé zkoumá a hledá potřebné vzory z kategorie vzorů chování. Jednotlivé uzly popisují metody a pole. Hrany pak jednotlivá volání mezi metodami. Podrobnosti viz. [23].

7.6.5 Design Pattern Detection By Template Matching

V této metodě se podobně jako v metodě Similarity scoring využívá matice. Principem této metody je výpočet na základě normalizované korelace dvou matic (systému a vzoru). Jádrem algoritmu je výpočet pomocí vzorce $CCn = \frac{\sum f(x)g(x)}{|f(x)||g(x)|}$,kde $f(x)$ a $g(x)$ jsou vektory, které popisují systém a vzor. Více podrobností o této metodě článek [24], kde je uvedený příklad na návrhovém vzoru Strategie.

V další části této práce se budeme výhradně zabývat metodou Similarity Scoring.

8 Metody detekce pomoci grafů

V této práci se zaměříme na detekci vzoru na základě převodu analyzovaného systému na orientovaný graf a následného porovnání s grafem, který reprezentuje návrhový vzor. Problematikou porovnání dvou grafů se zabývá tzv. graph matching (do češtiny volně přeloženo jako porovnávání grafů). V současné době existuje několik technik a přístupů k této problematice. Tento způsob detekce tak využívá strukturální popis analyzované aplikace a vzoru (tedy vztahů mezi jednotlivými objekty).

Hlavní rozdělení, které se v odborné literatuře uvádí ([9]), rozděluje možnosti detekce do takzvaných přesných a přibližných metod, které se dále dělí do dalších podkategorií(viz. obrázek 23).

8.1 Složitost

Složitost porovnání dvou grafu patří do kategorie NP-úplné. Je to dáno především tím, že úlohy řešení vedou ke komibnatorickým řešením, které často mají složitost $n!$. Důležité je, ale zmínit, že ne všechny porovnání grafu mají NP-úplnou složitost, jedná se především o grafy, které jsou planární.

8.2 Přesné metody

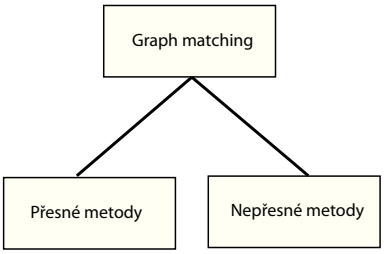
V angličtině se také používá termín exact graph matching. Tyto metody jsou založeny na izomorfizmu dvou grafů. Je tedy nutné, aby graf reprezentující vzor a zdrojový kód měly stejný počet uzlů. Protože často je graf reprezentující zdrojový kód větší než graf vzoru, je nutné přistoupit k vytváření podgrafů grafu zdrojového kódu o velikosti grafu vzoru. Pak je možné tyto grafy porovnat a zjistit, zdali jsou vzájemně izomorfní.

Vstupy algoritmu:

- Matice reprezentující systém
- Matice reprezentující vzor

Popis algoritmu:

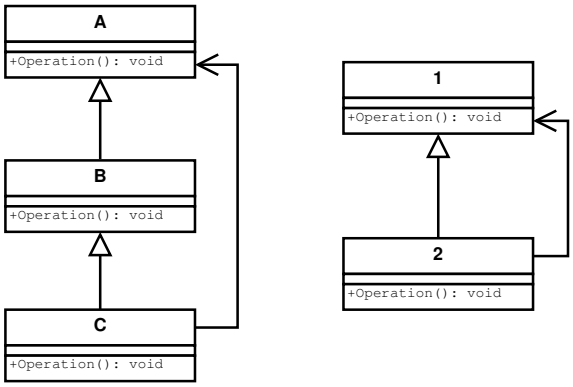
1. Vytvoření permutace množiny o počtu prvků, která je rovna stupni matice reprezentující systém
2. Vytvoření všech matic (ozn. $All(M)$) ze vstupní matice reprezentující analyzovaný systém se záměnou pořadí v řádcích a sloupcích
3. Vytvoření všech pod-matic (ozn. $SubAll(M)$) matic z $All(M)$ o stupni matice reprezentující vzor



Obrázek 23: Grafové rozdělení metod detekce

4. Porovnání všech matic ze $SubAll(M)$ maticí, která reprezentuje vzor.

Příklad:
Na jednoduchém příkladu si ukážeme jak tato metoda funguje.



Obrázek 24: Část systému a jednoduchý vzor

Nyní převedeme systém a vzor na matice sousednosti:

$$Sys_{asociace} = \begin{matrix} \\ A \\ B \\ C \end{matrix} \begin{matrix} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{matrix} \quad Patt_{asociace} = \begin{matrix} \\ 1 \\ 2 \end{matrix} \begin{matrix} \begin{matrix} 1 & 2 \end{matrix} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{matrix}$$

$$Sys_{generalizace} = \begin{matrix} \\ A \\ B \\ C \end{matrix} \begin{matrix} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{matrix} \quad Patt_{generalizace} = \begin{matrix} \\ 1 \\ 2 \end{matrix} \begin{matrix} \begin{matrix} 1 & 2 \end{matrix} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{matrix}$$

Pro další ukázku porovnání si zvolíme matice generalizace. Nyní je zapotřebí vytvořit všechny podmatice matice $Sys_{generalizace}$ o řádu matice $Patt_{generalizace}$. Nyní když máme vytvořené všechny podmatice matice $Sys_{generalizace}$, můžeme tyto matice porovnat s maticí $Patt_{generalizace}$. Když se matice sobě rovnají, je nalezena shoda a byl tak v matici $Sys_{generalizace}$ identifikován vzor. Tuto metodu musíme dále provést i s maticí, která reprezentuje asociace mezi třídami.

Výsledkem této metody je tedy seznam podmatic, které se rovnají matici sousednosti, která reprezentuje vzor. Z tohoto příkladu jsou to například tyto matice:

$$\begin{array}{c} \\ A \\ B \end{array}\!\!\begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ A \\ B \end{array}\!\!\begin{array}{c} \begin{array}{cc} A & C \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ C \\ B \end{array}\!\!\begin{array}{c} \begin{array}{cc} A & C \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array}$$

8.3 Přibližné metody

Tyto metody detekce řeší problém složitosti u přesných metod. Problémem přibližných metod je ten, jak již samotný název napovídá, že nedovedou přesně odpovědět na otázku, je-li návrhový vzor obsažen ve zdrojovém kódu. Výhodou těchto metod je i to, že graf vzoru a graf aplikace nemusí mít stejný počet uzlů. V této práci jsem se zaměřil na algoritmus „Similarity scoring“ a dříve než si ho detailně popíšeme, představíme si ještě jiné metody detekce.

8.3.1 Editační vzdálenost

Tato metoda je založena na principu počtu jednoduchých operací, pomocí kterých lze převést jeden graf na druhý. Operace jsou přidáni, odebráni a přejmenování hrany nebo uzlu. Jako výsledek je pak uváděné číslo, které znamená nejmenší počet operací, které byly použity k přechodu z jednoho grafu na druhý. Tato metoda se velice často používá při práci s textem a také s grafy. Více se o této metodě lze nají v [10].

8.3.2 Neuronové sítě

Pomocí neuronových sítí se dá řešit mnoho problémů spojená s teorií grafů a podobnosti dvou grafů. Vždy se na konkrétní problém volí konkrétní neuronová sít, se specifickou tréninkovou množinou. Typy problémů, které jsou spojeny s neuronovými sítěmi a grafy jsou např. autentifikace pomocí obličeje, dále pak předpovědi chování tropických cyklónů anebo zkoumání mozku. Více k této problematice lze najít v [11].

8.3.3 Fuzzy množiny

Pomocí teorie fuzzy množin lze řešit problematiku porovnání grafů. Fuzzy množiny slouží jako prostředek pro definici hran a uzlů grafu a následné aplikace na analyzovaný

graf. Typickým použitím této metody je k detekci vzdálenosti v obrazech. Více se o této problematice můžeme dočíst například [15]

8.3.4 Similarity scoring

Tento algoritmus jsem si vybral pro implementaci z kategorie přibližných metod. Jeho principem je výpočet podobnostní matice, která udává, jak moc je návrhový vzor obsažen ve zdrojovém kódu. Vychází ze vzorce, který vypočte podobnostní matici na základě dvou různých grafů (více viz. [12]). Algoritmus je založen na iterativním způsobu výpočtu podobnostní matice. Dokud se dvě předchozí matice neliší o předem nadefinovanou hodnotu, výpočet pokračuje dál.

Vstupem výpočtu jsou dvě matice A a B, které reprezentují zdrojový kód a návrhový vzor. Podobnostní matice S velikosti $n_a \times n_b$ je definována prvky s_{ij}, které reprezentují, jak moc uzel j z grafu G_a a tedy matice sousednosti A, je podobný uzlu i z grafu G_b tedy matice B. Hodnota v s_{ij} je nazývána podobnostní skóre.

Popis algoritmu:

1. $Z_0 = 1$, v matici Z jsou uloženy dočasné výsledky výpočtu. Na začátku jsou všechny prvky matice nastaveny na 1
2. Dokud matice Z nebude konvergovat, opakuje se výpočet pomocí tohoto vzorce:

$$Z_{k+1} = \frac{BZ_kA^T + B^TZ_kA}{\|BZ_kA^T + B^TZ_kA\|_1}$$

3. Výsledná matice S je poslední matice Z_k

Vysvětlivky:
A, B jsou matice sousednosti grafů G_a a G_b
$\|.\|_1$ je výpočet 1-normy.

Výpočet první normy:
První norma matice se vypočte pomoci vzorce:

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$

Vzorec vypočte maximální hodnotu sloupců zadané matice.

Konvergence matice:
Výpočet podobnostní matice končí ve chvíli, kdy matice konverguje. K tomu, aby se určilo zdali matice konverguje existuje několik přístupů. Ve své práci jsem si zvolil konvergenci matice na základě výpočtu normy matice a následném porovnání s hodnotou normy v předchozím kroku. Jako nejúčinnější normou se jeví euklidovská norma, která je předepsána tímto vzorcem:

$$\|A\|_\infty = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2}$$

Jako další normy by se daly použít normy řádkové nebo sloupcové.

Příklad:
Jako ukázku výpočtu pomocí tohoto algoritmu, použijeme již výše uvedený příklad u přesných metod. Po převodu malého systému (viz. obrázek 24) a návrhového vzoru, můžeme přejít k samotnému výpočtu podobnostní matice.

Nejprve vypočteme matici, která reprezentuje asociace v programu a v návrhovém vzoru:

Vstup:

- Matice asociace reprezentující aplikaci
- Matice asociace reprezentující vzor

Výstup:

$$S_{asociace} = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$$

Dále provedeme výpočet s maticemi reprezentujícími generalizaci:

Vstup:

- Matice generalizace reprezentující aplikaci
- Matice generalizace reprezentující vzor

Výstup:

$$S_{generalizace} = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} 0.5 & 0 \\ 0.5 & 0.5 \\ 0 & 0.5 \end{pmatrix} \end{array}$$

Nyní máme k dispozici dvě matice, které reprezentují podobnostní skóre za použití matic asociací a generalizací. K výsledné podobnostní matici se dostaneme tak, že matice sečteme a výslednou matici normalizujeme. Matice, kterou výsledek normalizujeme má

na hlavní diagonále hodnoty $1/k$. Číslo k je rovno počtu matic ve kterých se vzor popisoval, tzn. v našem případě $k = 2$.

Výsledná podobnostní matice S:

$$S = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} 0.75 & 0 \\ 0.25 & 0.25 \\ 0 & 0.75 \end{pmatrix} \end{array}$$

Výsledná matice S se tedy dá číst tak, že jednotlivé řádky jsou třídy reprezentující analyzovaný systém a sloupce jsou třídy vzoru. Když tedy výslednou matici budeme analyzovat, z matice vyplývá, že existuje silná podobnost mezi třídami $(A, 1)$ a $(C, 2)$. Podle obrázku 24 jsou tedy ukázkový systém a ukázkový vzor vzájemně podobné.

Jako poslední část této kapitoly si uvedeme vlastní implementaci algoritmu Similarity scoring, včetně metody, která zajišťuje zastavení výpočtu.

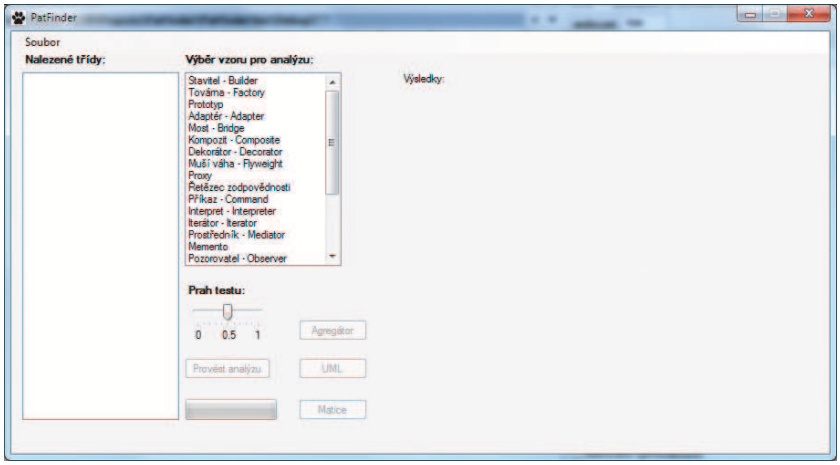
```
public double[,] CalculationSimilarityScore(double[,]
MatrixCode, double[,] MatrixPattern)
{
    int x = MatrixCode.GetLength(0);
    int y = MatrixPattern.GetLength(0);
    double[,] MatrixResult = MatrixFill (1, y, x);
    double[,] TempMatrix = new double[y, x];
    double[,] MatrixTemp = null;
    double[,] MatrixTemp1 = null;
    double[,] MatrixTemp2 = null;
    bool isConvergate = false;
    while (!isConvergate)
    {
        MatrixTemp1 = math.MatrixMultiple(math.MatrixMultiple(
        MatrixPattern, MatrixResult),
        math.MatrixTranspone(MatrixCode));

        MatrixTemp2 = math.MatrixMultiple(math.MatrixMultiple(
        math.MatrixTranspone(MatrixPattern), MatrixResult),
        MatrixCode);

        MatrixTemp = math.MatrixAdd(MatrixTemp1, MatrixTemp2);
        MatrixResult = math.MatrixDivide(MatrixTemp, math._1norm(
        MatrixTemp));
        isConvergate = Convergence(MatrixResult, TempMatrix);
        TempMatrix = MatrixResult;
    }
    return math.MatrixTranspone(MatrixResult);
}

public bool Convergence(double[,] A, double[,] B)
```

```
{
     iii ++;
    double euclid_a = 0;
    for (int i = 0; i < A.GetLength(0); i++)
    {
        for (int j = 0; j < A.GetLength(1); j++)
        {
            euclid_a += Math.Pow(A[i, j], 2);
        }
    }
    euclid_a = Math.Sqrt(euclid_a);
    double euclid_b = 0;
    for (int i = 0; i < B.GetLength(0); i++)
    {
        for (int j = 0; j < B.GetLength(1); j++)
        {
            euclid_b += Math.Pow(B[i, j], 2);
        }
    }
    euclid_b = Math.Sqrt(euclid_b);
    if ((euclid_b + euclid_a) == LAST_VALUE)
    {
        return true;
    }
    else
    {
        LAST_VALUE = euclid_b + euclid_a;
        return false;
    }
}
```

Výpis 1: Algoritmus Similarity Scoring

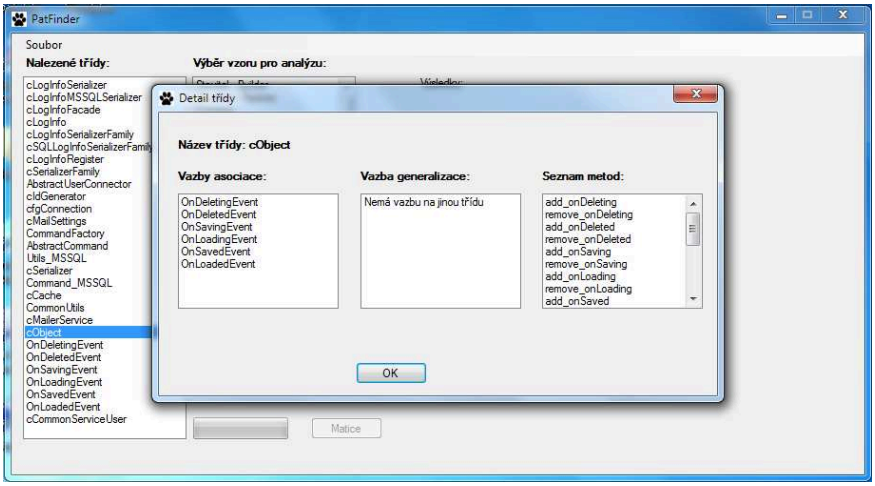
9 Uživatelská příručka

Abychom mohli vybrané metody ověřit pomocí experimentů, vytvořil jsem aplikaci PatFinder, která implementuje metodu Similarity Scoring. Jedná se o jednoduchou aplikaci, která obsahuje několik vzorů vhodných pro experimenty a dále nabízí základní možnosti práce s načtenou komponentou. Výsledky detekčních metod jsou pak jednoduše zobrazeny. Obsahem uživatelské příručky je seznámit čtenáře s vlastním ovládáním aplikace a jejími možnostmi. Aplikace PatFinder je vytvořená jako klasická desktopová aplikace v prostředí Microsoft Windows. K jejímu spuštění je zapotřebí mít nainstalovaný .NET framework 2.0 nebo vyšší. Aplikace nevyžaduje instalaci, proto je možné aplikaci spouštět přímo. Po spuštění aplikace se zobrazí okno (viz. obrázek 25), v tuto chvíli můžeme začít s aplikací pracovat. Aplikace je rozdělena na tři záložky. V první záložce lze načíst zdrojový kód a dále nastavit návrhový vzor k porovnání.



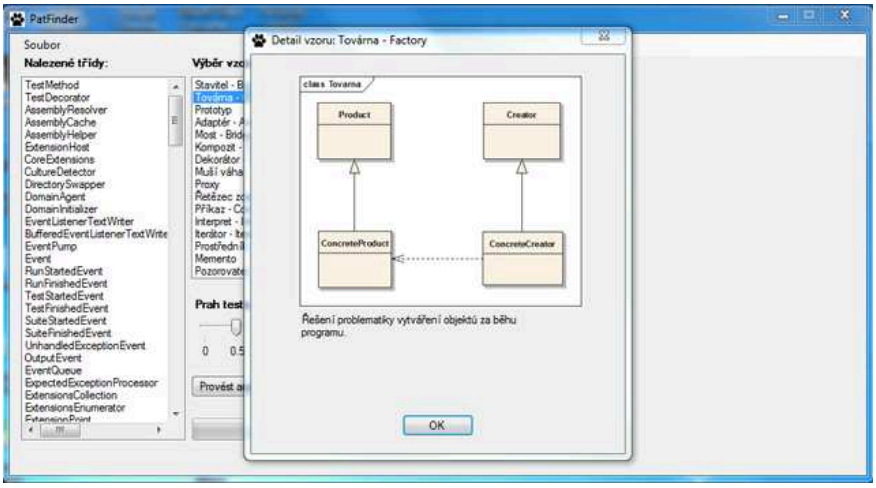
Obrázek 25: Úvodní obrazovka aplikace

Po načtení zkompilovaného zdrojového kódu, se zobrazí seznam tříd, které byly nalezeny. Mezi třídami je možné listovat a prohlížet si tak jejich vazby asociace a generalizace a také je možné podívat se na seznam metod (viz. obrázek 26).



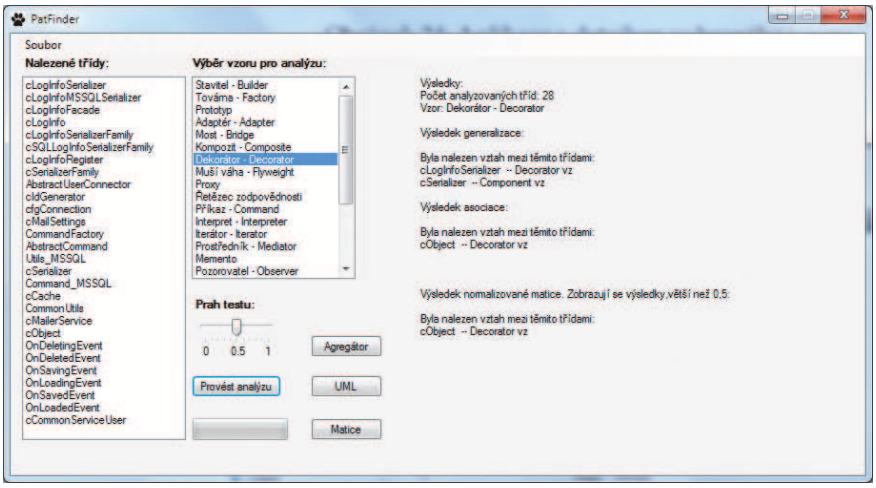
Obrázek 26: Aplikace po načtení komponenty

Uživatel má možnost vybrat si návrhový vzor, na kterém chce provést analýzu a po dvojkliku na název vzoru se mu zobrazí schéma vzoru a krátky popis. Viz obrázek 27.



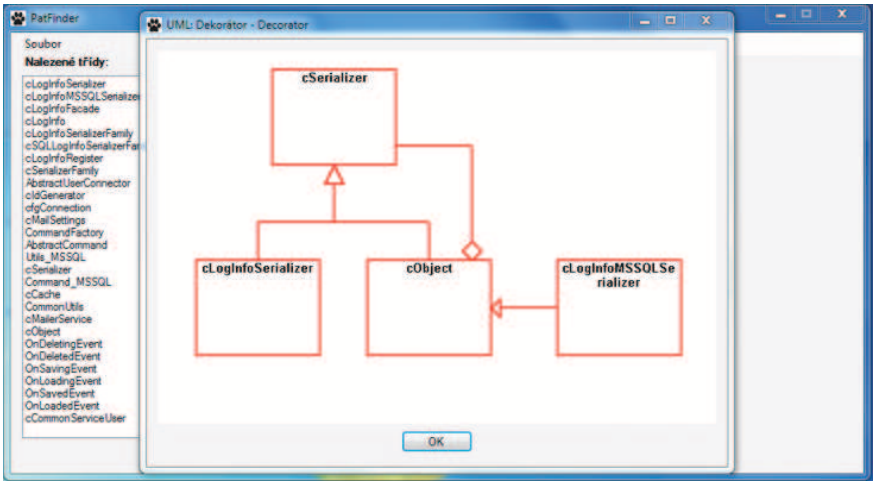
Obrázek 27: Aplikace s detailem vybraného vzoru

Jakmile je načtená analyzovaná komponenta a je vybrán návrhový vzor. Aktivuje se tlačítko „Provést analýzu“. Na konci výpočtu je pak v pravé části aplikace - označenou textem „Výsledky“, zobrazen výsledek analýzy (viz. obrázek 28).



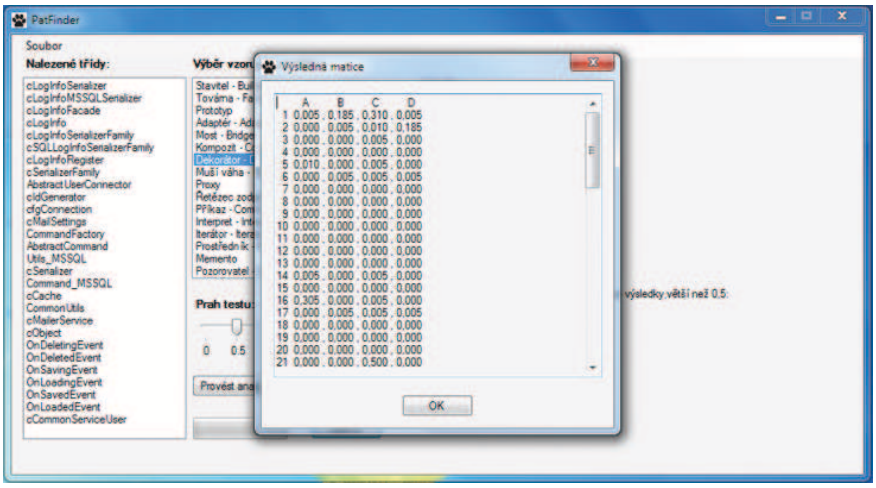
Obrázek 28: Výsledky analýzy

Dále je možné zobrazit si grafickou podobu návrhového vzoru, který byl analyzován. Místo názvů tříd návrhového vzoru jsou ale zobrazeny názvy tříd, které se nejvíce podobají jednotlivým třídám vzoru(viz. obrázek 29).



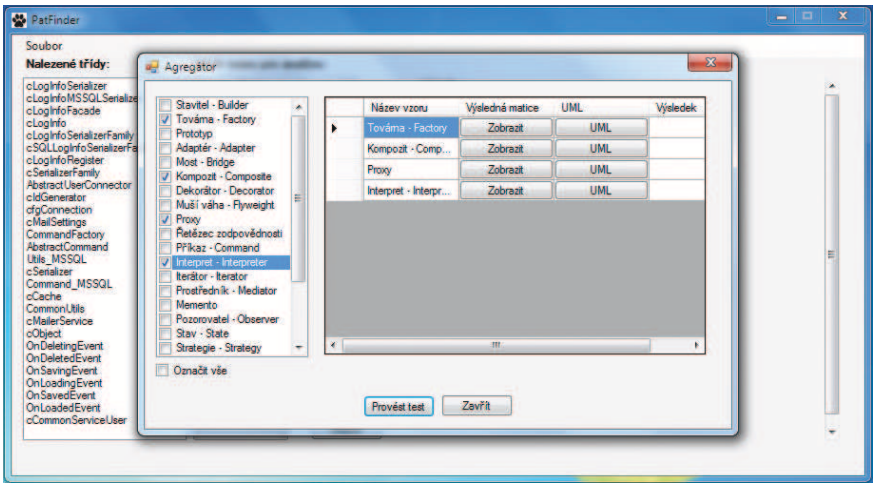
Obrázek 29: Výsledky analýzy - zobrazení vzoru

Také je možné si zobrazit výslednou matici výpočtu. Obsah okna je možné si zkopírovat a tak dále s výsledkem pracovat(viz. obrázek 30).



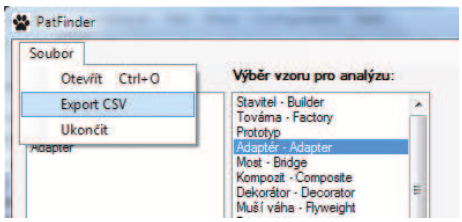
Obrázek 30: Výsledky analýzy - zobrazení matice

V aplikaci je dále možnost využit vyhledávání prostřednictvím agregátoru, kde si lze vybrat více vzorů pro analýzu a následně se provede výraz pro jednotlivé zvolené vzory. Více viz. obrázek 31



Obrázek 31: Analýza pomocí agregátoru

Poslední možností je uložení výsledné matice do standartního formátu CSV (viz obrázek 32)



Obrázek 32: Export do formátu CSV

10 Popis implementace

V této části práce se budeme zabývat popisem praktické části diplomové práce. Aplikace, kterou jsem vytvořil, se jmenuje PatFinder a aplikuje vybranou metodu similarity scoring. Nejprve si napíšeme něco o analýze požadavků, pak se zaměříme na návrh aplikace a klíčové funkce. Dále pak se seznámíme s uživatelskou příručkou a nakonec, v programátorské části, popíšeme jednotlivé metody aplikace. Na obrázku 33 je znázorněn datový tok analýzy vybrané komponenty.

10.1 Specifikace požadavků

Úkolem aplikace je načíst překompilovaný program v prostředí .NET framework, dále pak pomocí reflexe detekovat všechny třídy a rozhraní a na základě vazeb typu asociace a generalizace sestavit orientovaný graf, který se v aplikaci reprezentuje pomocí matice sousednosti. Dále je v aplikaci umožněn výběr návrhového vzoru, je možnost nahlédnout na jeho strukturu a následně provést analýzu na základě algoritmu similarity scoring. Výsledky jsou pak zobrazeny tak, že se vypisují názvy tříd, mezi kterými byla nalezena podobnost. V aplikaci je možné nastavovat práh, přes který se budou podobnosti vypisovat.

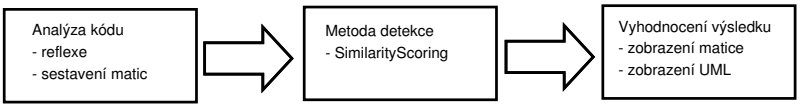
10.2 Návrh aplikace

Návrhem aplikace se myslí především diagram tříd a diagram vrstev. Protože samotná aplikace není příliš složitá, stačila jednoduchá analýza, která ve svém výsledku navrhuje vytvořit dvou-vrstvou aplikaci, kde uživatelské rozhraní nabízí základní ovládání ze strany uživatel, tak jak je standardně zvykem a pak business vrstvu, která se stará jak o vlastní logiku programu, tak i o vstupy a výstupy aplikace.

10.2.1 Diagram tříd

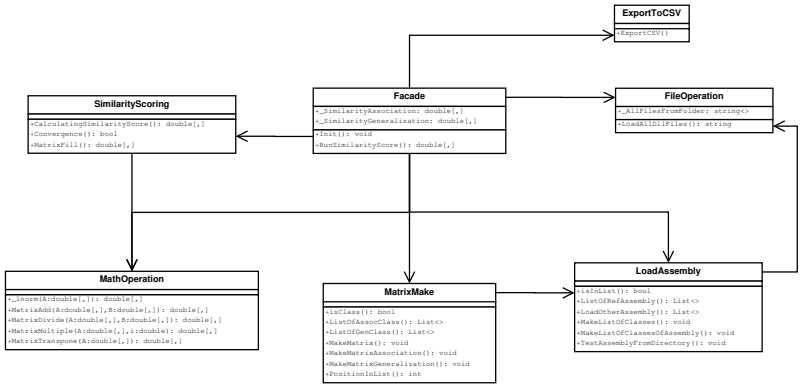
V této podkapitole si vyjmenuje a popíšeme třídy, které se nacházejí v tzv. business vrstvě aplikace a které mají za úkol načíst zdrojový kód a následně na něm provést detekci vybraného vzoru.
V aplikaci jsou tyto třídy:

- **Facade** - třída, která umožňuje vstup do logiky aplikace
- **FileOperation** - tato třída zajišťuje operace se soubory komponent
- **LoadAssembly** - v této třídě se analyzují a načítají jednotlivé assembly (komponenty)
- **MathOperation** - třída s metodami pro počítání s maticemi
- **MatrixMake** - pomocí této třídy se vytváří matice sousednosti



Obrázek 33: Datový tok

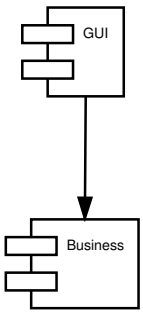
- **SimilarityScoring** - tato třída obsahuje metody pro výpočet algoritmu similarity scoring
- **PatternAnalyze** - tato třída zpracovává výsledky výpočtů
- **ExportToCSV** - třída k exportu do formátu CSV



Obrázek 34: Diagram tříd

10.2.2 Diagram vrstev

Jak již bylo napsáno na úvod této kapitoly, jedná se o dvou-vrstvou aplikaci. Proto tzv. business vrstva kromě vlastní logiky aplikace pracuje i se vstupy a výstupy.



Obrázek 35: Diagram vrstev

10.3 Klíčové funkce

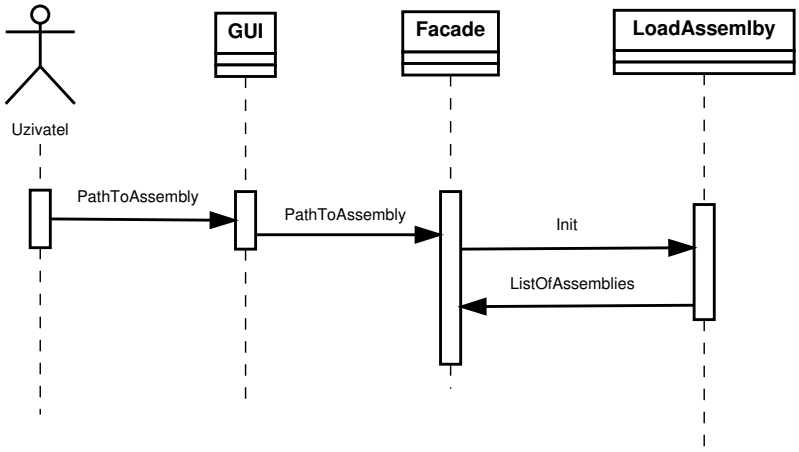
V aplikaci PatFinder můžeme najít několik důležitých funkcí, mezi ty klíčové bychom mohli zařadit tyto tři:

- Načtení assembly
- Sestavení matic asociace a generalizace mezi třídami a rozhraními
- Detekce vzoru na základě similarity scoringu

Ke každé klíčové funkci si uvedeme sekvenční diagram, který nám znázorňují časové uspořádání událostí mezi jednotlivými objekty.

10.3.1 Načtení assembly

Pomocí této klíčové funkce se načtou všechny potřebné assembly, které dále poskytnou seznamy tříd k sestavení matic. Sekvenční diagram je uveden zde 36.



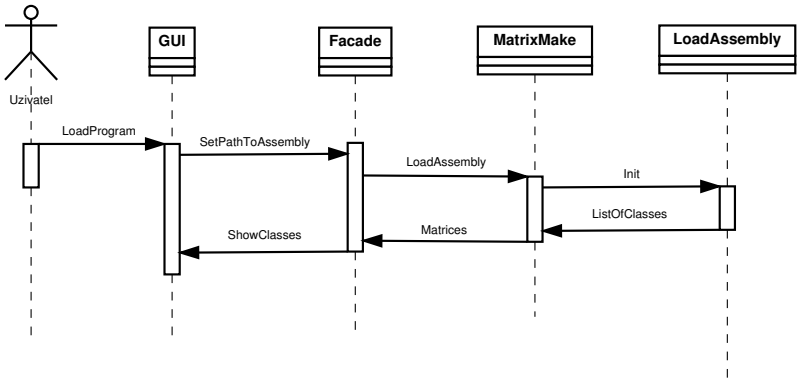
Obrázek 36: Načtení assembly

10.3.2 Sestavení matic asociace a generalizace

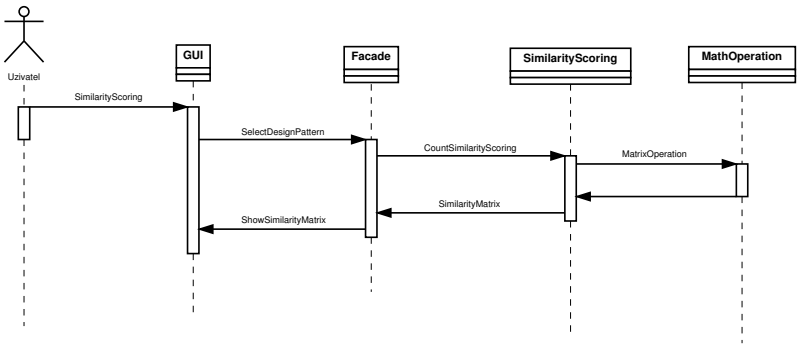
V této klíčové funkci se načtou jednotlivé assembly s překompilovanými zdrojovými kódy, následně vytvoří seznam tříd a rozhraní a vytvoří dvě matice, které reprezentují asociaci a generalizaci mezi třídami. Sekvenční diagram je uveden zde 37.

10.3.3 Detekce vzoru na základě similarity scoring

V této poslední klíčové funkci se realizuje detekce vzoru za použití nepřesné metody a to algoritmu similarity scoring. Sekvenční diagram je uveden zde 38.



Obrázek 37: Sestavení matic asociace a generalizace



Obrázek 38: Detekce vzoru na základě similarity scoring

10.4 Slovní popis implementace

První část aplikace (první modul) má na starosti načtení analyzovaného programu a následné sestavení matic, které reprezentují asociaci a generalizaci v analyzované aplikaci. K tomu abych načetl analyzovanou aplikaci využívám v rámci .NET frameworku technologii reflexe. Tato technologie umožňuje zpětnou analýzu a práci s aplikacemi, které byly v rámci .NET frameworku zkompilovány. Postup při sestavení matic je takový, že se načte dll nebo exe soubor jako datový typ Assembly,který dále nabízí seznam všech objektů a vztahů mezi nimi.
Načtení assembly se provádí pomocí třídy Assembly:

```
Assembly assembly = Assembly.LoadFrom(path);
```

V tuto chvíli jsou v proměnné assembly načtené metainformace k analyzované komponentě. Protože jsou většinou vyvíjené aplikace rozděleny do více komponent (např. jedna realizuje logiku aplikace, druhá například přístup k datům) je vhodné k analýze komponenty připojit i ostatní komponenty, které logicky do aplikace patří. K této informaci

(seznamu referencí) je možné se dostat pomocí metody `GetReferencedAssemblies()`. Metoda nám tedy vrací seznam referencí a následně je adresář, kde se samotná komponenta nachází prozkoumán, zdali neobsahuje komponenty, které patří do referencí. Jsou li nalezený, jsou připojeny k analýze a k sestavení matic.

Jakmile je sestaven seznam všech možných komponent k analýze, je vytvořen seznam jednotlivých tříd, které jsou obsaženy v metainformacích od jednotlivých komponent. K těmto informacím se dostaneme pomocí metody `GetTypes()`, která vrací podrobné informace o všech typech, které jsou v assembly zastoupeny. Jedním z typů, které jsou v metainformacích obsaženy jsou jednotlivé třídy, ze kterých je komponenta složena. Třídy, které jsou zařazeny do seznamu, jsou testovaný vlastnostmi `isClass`, `isInterface` a `isAbstract` a to z toho důvodu, že v metainformacích se nacházejí i třídy, které nejsou přímo součástí systému, ale kompilátor je tam z důvodů optimalizace přidává.

Ve chvíli, kdy je sestaven kompletní seznam tříd, je možné vytvořit matice, které reprezentují vztahy generalizace a asociace mezi třídami.

Sestavení matice, která reprezentuje generalizaci – dědičnost slouží vlastnost `BaseType`. Tato vlastnost v sobě zanechává název třídy, která je nadřazená analyzované třídě. Vždy se otestuje, zdali nadřazená třída je obsažena v seznamu tříd, pokud je, pak je mezi analyzovanou třídou a nadřazenou třídou vytvořena vazba.

K vytvoření matice reprezentující asociace mezi třídami je zapotřebí si ke každé analyzované třídě načíst pomocí metody `GetFields()` informace, které popisují veřejné informace v analyzované třídě. Jedna z těchto informací je seznam tříd, na které se analyzovaná třída odkazuje. Tato vlastnost nám tedy umožní sestavit matice asociací. Nutno podotknout, že při sestavování matice může dojít k situaci, kdy není asociace mezi třídami detekována. To je způsobeno špatnou deklarací. Více informací k technologii reflection je možná najít např. [25].

10.5 Popisy metod

V této kapitole si přiblížíme nejdůležitější metody, které se v aplikaci vyskytují. Jejich rozdělení je podle jednotlivých tříd, do kterých patří a dále abecedně seřazené.

10.5.1 FileOpertion

Tato třída zajišťuje ověření a načtení dll nebo exe souboru.

- `public void LoadAllDllFiles(string pFolder)` - načte všechny dll soubory ze zadaného adresáře a všech podadresářů

10.5.2 LoadAssembly

V tét třídě se načtou jednotlivá sestavení a vytvoří se seznam jednotlivých tříd analyzovaného systému.

- `public List<string> ListOfRefAssembly()` - načte všechny assembly, na které se anaylzovaná assembly odkazuje

- `public void TestAssemblyFromDirectory()` - otestuje přítomnost assembly v zadaném a adresáři
- `public bool isInList(string assName)` - test na přítomnost assembly v seznamu
- `public Assembly LoadOtherAssembly(string path)` - načtení assembly
- `public void MakeListOfClasses()` - vytvoření seznamu tříd

10.5.3 Metody třídy MathOperation

Třída poskytuje matematické operace nad maticemi.

- `Public double _1norm(double[,] A)` - výpočet první normy matice
- `Public double[,] MatrixAdd(double[,] A, double[,] B)` - metoda na součet dvou matic
- `Public double[,] MatrixDivide(double[,] A, double[,] B)` - dělení matice konstantou
- `Public double[,] MatrixMultiple(double[,] A, double[,] B)` - násobení dvou matic
- `Public double[,] MatrixTranspone(double[,] A)` - transpozice matice

10.5.4 Metody třídy MatrixMake

V této třídě se vytváří matice asociace a generalizace

- `public void MakeMatrix()` - základní inicializace proměnných a volání jednotlivých metod pro sestavení matic asociace a dědičnosti
- `private void MakeMatrixAssociation(Type[] types)` - vytvoří matici sousednosti, ve které jsou reprezentovány vazby typu asociace mezi třídami
- `private void private void MakeMatrixGeneraliztion(Type[] types)` - vytvoří matici sousednosti, ve které jsou reprezentovány vazby typu generalizace mezi třídami
- `Private bool isClass(string cname)` - v této metodě se otestuje, je-li zadaná třída v seznamu tříd
- `Private int PositionInList(string cname)` - metoda, která určí pozici matice v rámci matice
- `public List<string> ListOfAssocClass(string cname)` - pomocná metoda pro vazby typu asociace

- `public List<string> ListOfGenClass(string cname)` - pomocná metoda pro vazby typu generalizace

10.5.5 Metody třídy SimilarityScoring

Ve třídě se realizuje výpočet pomocí metody Similarity scoring.

- `Public double[,] CalculationSimilarityScore(double[,] MatrixCode, double[,] MatrixPattern)` - metoda na vlastní výpočet similarity score
- `Public bool Convergence(double[,] A, double[,] B)` - metoda výpočtu na konvergenci matice
- `Public double[,] MatrixFill(double what, int x, int y)` - pomocná metoda před-vyplnění matice

10.5.6 Metody třídy PatternAnalyze

Tato třída analyzuje výsledné matice a sestavuje je do uživatelsky přívětivého vzhledu.

- `public List<int[,]> Analyze()` - metoda, která analyzuje výsledky po výpočtu podobnostní matice
- `public bool isNull(double[,] mat)` - test nulové matice

10.5.7 Metody třídy ExportToCSV

V této třídě se výsledná matice exportuje do formátu CSV.

- `public void ExportCSV(double[,] arr, List<String> list, string path)` - metoda na export CSV

11 Experimenty

Součásti této práce jsou experimenty, které byly provedeny v aplikaci PatFinder. Experimentem tedy myslíme, pokus na vstupních datech (komponentách), které nám poskytnou výsledky o tom, je-li nějaký vzor obsažený v komponentě nebo ne. Máme tedy k dispozici komponenty a dále máme možnost pomocí reflexe sestavit vztahy mezi třídami v komponentě. Tyto vstupní informace, tedy matice asociací a generalizací jsou vstupem pro přesné a přibližné metody.

V této části práce si popíšeme pokusy, které byly v rámci aplikace PathFinder provedeny na několika typech vstupních dat. Vstupní data bychom mohli rozdělit do dvou kategorií:

- Pozitivní příklady
- Negativní příklady

11.1 Pozitivní příklady

Těmito vstupními daty jsou myšlená takové data, u kterých víme, že návrhový vzor je obsažen ve zdrojovém kódu programu. Těmito daty můžeme tak prokázat spolehlivost detekčních metod a posoudit jejich kvalitu. Do pozitivních dat tedy zvolíme data, které jsou speciálně vytvořena pro tuto aplikaci, a tedy to nejsou data reálných programů.

Vytvořili jsme tedy sadu testovacích vstupních dat, které jsme poté analyzovali. Jedná se o velice jednoduchou komponentu, která je již uvedena v příkladech u jednotlivých metod.

11.2 Negativní případy

Negativními případy jsou myšlená taková vstupní data, o kterých víme, že v nich nejsou vzory obsaženy. Použité metody detekce by měly prokázat, že vzory nejsou detekovány. Jak se ale ukáže níže, ne vždy se ale tato domněnka potvrdí.

11.3 Vlastní experimenty

Experimenty jsou tedy provedeny jak na komponentách, které jsou vytvořeny přímo pro ověření platnosti metody detekce, tak na komponentách reálných aplikací. Všechny testované komponenty jsou k nalezení na přiloženém CD v adresáři `DATA`.

11.3.1 Experiment č. 1

První experiment byl proveden na testovací komponentě, která byla pro tuto potřebu vytvořena. Jedná se o malou část systému, na které již výše byla demonstrována metoda detekce (24). Vzor je zde reprezentován malým vzorem, který se jmenuje element.

Výsledek analýzy je uveden v tabulce:
Vyhodnocení:

Analyzovaný systém	Vzor element
Class1	A
Class3	B

Tabulka 3: Experiment č. 1

Tímto experimentem se na jednoduché komponentě a jednoduchém vzoru demonstrovala ukázka toho, že obě metody našly vztah mezi komponentou a vzorem.

11.3.2 Experiment č. 2

V tomto experimentu jsem se pokusil ověřit metodu na návrhovém vzoru Adaptér. Vzor adaptér popisuje řešení problému konverze rozhraní třídy na rozhraní jiné třídy (viz. obrázek 6).

Výsledek analýzy na provedene komponentě, lze shrnout v tabulce:

Analyzovaný systém	Vzor adaptér
Výsledek generalizace	
Adaptee	Target
Adapter	Adapter
Výsledek asociace	
Adapter	Target
Adapter	Adaptee
Client	Client
Client	Adapter
Normalizovaná matice	
Adaptee	Target
Adapter	Adapter

Tabulka 4: Experiment č. 2

Vyhodnocení:
Z tabulky je patrná sílná podobnost mezi třídami `Adaptee` z analyzované komponenty a třídou `Target` z návrhového vzoru. Dále je vidět silná podobnost mezi třídami `Adapter` a `Adapter`.

11.3.3 Experiment č. 3

Další experiment je provedeny na návrhovém vzoru Kompozit (viz. obrázek 8). Tento vzor řeší problematiku uspořádání objektů do stromové struktury a následný přístup k

objektům nebo skupině objektů. K testování byla použita aplikace, která skládá grafické objekty (čára,kruh) do složitějších objektů, které jsou z těchto objektů poskládány.

Výsledky analýzy jsou shrnuty v tabulce.

Analyzovaný systém	**Vzor kompozit**
Výsledek generalizace	
DrawingElement	Component
Výsledek asociace	
MainApp	Client
MainApp	Composite
PrimitiveElement	Component
CompositeElement	Component
Normalizovaná matice	
DrawingElement	Component

Tabulka 5: Experiment č. 3

Vyhodnocení:
Z tabulky (**??**), ve které jsou zobrazeny výsledky je patrné, že mezi třídami, které reprezentují v analyzované komponentě třídy návrhového vzoru je silná podobnost. Podobnost je nejsliněji vidět u tříd `DrawingElement` a `Component`.

11.3.4 Experiment č. 4

V dalším experimentu je provedena analýza na komponentě, která má za úkol logovat provoz v rámci systému. Na komponentě byla provedena série testů s různými vzory. O komponentě je známo, že v ní byl použit vzor Dekorátor (viz. obrázek 7). Tento vzor slouží k přidáváni funkčnosti během běhu objektu.

Výsledky analýzy jsou shrnuty v tabulce.

Analyzovaný systém	**Vzor dekorátor**
Výsledek generalizace	
cLogInfoSerializer	Decorator
cSerializer	Component
Výsledek asociace	
cObject	Decorator
Normalizovaná matice	
cObject	Decorator

Tabulka 6: Experiment č. 4

Vyhodnocení:
Mezi třídami `cObject` a `Decorator` byla detekována silná podobnost. Tato podobnost je dána tím, že `cObject` v systému je navrhnuta jako třída vzoru `Decorator`.

11.3.5 Experiment č. 5

V tomto experimentu byla metoda detekce vyzkoušena na návrhovém vzoru `Továrna` (viz. obrázek 3). testovaná komponenta byla pro tento experiment speciálně vytvořená. Výsledky analýzy jsou shrnuty v tabulce.

Analyzovaný systém	**Vzor tovární metoda**
Výsledek generalizace	
Product	Product
Product	Creator
Creator	Product
Creator	Creator
Výsledek asociace	
nic nenalezeno	nic nenalezeno
Normalizovaná matice	
Product	Product
Product	Creator
Creator	Product
Creator	Creator

Tabulka 7: Experiment č. 5

Vyhodnocení:
U tohoto experimentu byla nalezena silná vazba mezi třídami uvedenými v tabulce. Vazba byla detekována pouze u generalizace, ale již nebyla detekována u asociace. Tento experiment ukázal, že metoda funguje, pokud jsou dobře sestaveny matice asociace a generalizace. Jedním z problému při využití reflexe je ten, že nedokáže dostatečně dobře detekovat vazby typu asociace. Experiment dále ukázal, že metoda již nedokáže rozlišit to, které třídy k sobě patří a které ne. Patrné je to z tabulky u tříd `Product` a `Creator`.

11.4 Shrnutí experimentů

Z provedených experimentů na různých vstupních datech jsem došel k závěru, že metoda nejlépe detekuje tyto návrhové vzory:

- Adaptér
- Příkaz
- Dekorátor

- Kompozit
- Stav
- Pozorovatel
- Tovární metoda
- Šablonová metoda
- Most

Obdobná skupina návrhových vzorů, které metoda Similarity scoring detekovala je uvedena v článku: A review of design pattern mining techniques (více viz [26]). Tento článek shrnuje jednotlivé metody detekce a nalezené vzory. Metoda Similarity scoring se tak nedá použít pro detekci všech návrhových vzorů v rámci rozdělení GoF a musí tak být doplněna ještě další metodou. Důvodem proč nelze detekovat všechny návrhové vzory je ten, že metoda se zabývá syntaktickým popisem vzoru a analyzovaného systému. Některé vzory (především vzory chování) ale ke svému správnému použití potřebují využít metody, které mají implementovány. Z tohoto důvodu metoda Similarity scoring nepokrývá všechny návrhové vzory rozdělené dle GoF.

12 Závěr

V této diplomové práci jsem se pokusil popsat metody a přístupy k vyhledávání návrhových vzorů v rámci .NET frameworku.

Vybral jsem si metodu Similarity scoring, kterou jsem teoreticky popsal a dále jsem je implementoval a vytvořil aplikaci PatFinder, která metodu implementuje. Výsledkem této práce jsou provedené experimenty, pomoci kterých jsem ověřil, že metoda funguje a dává zajímavé výsledky o analyzovaných komponentách. Metoda prokázala, že některé návrhové vzory nalezne a je pak jen na expertovi, zdali nalezené podobnosti odpovídají i tomu, jak je systém navržen a realizován. Aby bylo možné dosáhnout lepších výsledků bylo by vhodné mít ještě nějakou jinou metodu, která by se ale na návrhové vzory dívala z jiného úhlu pohledu a ne jen z pohledu syntaktického.

Výsledky této práce mi poslouží k dalšímu zkoumání v této oblasti a k vytvoření optimalizačního postupu při vytváření matic sousednosti a následné analýzeatic sousednosti a následné analýze.

13 Conclusions

I tried to describe the methods and approaches to the design patterns detection in .NET Framework in my diploma thesis.

I chose similarity score method which I described, after that I implemented it and developed application called PatFinder. The results of my thesis are the experiments due to them I have verified that the chosen method works and give the interesting results about the analyzed components. Unambiguously is confirmed that if the design pattern contained in a component, the methods detected it. The method showed that some design patterns are found and then it´s up to the expert if the similarities are consistent with how the system is designed and implemented. In order to achieve better results it would be appropriate for some other method would be to design patterns, but looked from another perspective and not just in terms of syntax.

The results of the thesis will allow me another research in this area and creation of an optimization procedure in creating of anadjacency matrix and analysis.

14 Literatura

[1] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, Spyros T. Halkidis: *Design pattern detection using similarity scoring*, IEEE Transactions on Software Engineering, 2006

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994

[3] Arnošt Večerka: *Grafy a grafové algoritmy*, Olomouc, 2007

[4] Daniel Hort, Jiří Rachůnek : *Algebra I*, VUP, Olomouc, 2003

[5] Judith Bishop: *C# 3.0 Design Patterns*, O'REILLY, 2007

[6] Jim Arlow, Ils Neustadt: *UML 2 a unifikovaný proces vývoje aplikací*, Computer Press, 2007

[7] Simon Robinson,K.Scott Allen,Ollie Cornes,Jay Glynn,Zach Greenvoss, Burton Harvey,Christian Nagel,Morgan Skinner,Karli Watson: *C# Programujeme profesionálně*, Computer Press, 2003

[8] Ondřej Lehečka: *PATTRON patterns online*, 2005

[9] Endika Bengoetxea: *Inexact Graph Matching Using Estimation of Distribution Algorithms*, 2002

[10] Kaspar Riesen: *Bipartite Graph Matching for Computing the Edit Distance of Graphs* , Bern, 2007

[11] P.N. Suganthan, E.K. Teoh, D.P. Mital: *Pattern recognition by homomorphic graph matching using Hopfield neural networks* , Singapore, 1995

[12] V.D.Blonde, A. Gajaro, M. Heymans, P.Senellart, P. Van Dooren: *A measure of similarity between graph vertices:Applications to synonym extraction and web searching*, SIAM Rev., vol. 46, no. 4, pp. 647-666, 2004

[13] Craig Larman: *Applying UML and Patterns*, Prentice Hall, 2009

[14] Vondrák I., Kožusznik J., Ochodková E.: *Metody specifikace softwarových systémů*, VŠB, 2006

[15] Isabelle Bloch: *Fuzzy Relative Position Between Objects in Image Processing: A Morphological Approach*, IEEE Transactions on Pattern Analysis and Machine Intelligence , 1999

[16] Fowler M.: *Patterns of enterprise application architecture*, Addison Wesley, 2003

[17] Wikipedia: *Security Patterns*

[18] Kraval Ilja, RNDr.: *Vzory, které byste měli znát*,Object conculting s.r.o.,2007

[19] Data & Object Factory, LLC.: *http://www.dofactory.com/Patterns/Patterns.aspx*, Data & Object Factory, LLC.

[20] Y. Guéhéneuc, H. Sahraoui, a F. Zaidi: *Fingerprinting design patterns.*, Proceedings of the 11^{th} Working Conference on Reverse Engineering (WCRE), 2004

[21] R. Ferenc, A. Beszedes, L. Fulop, a J. Lele: *Design pattern mining enhanced by machine learning.*, In Proceedings of the 21^{st} IEEE International Conference on Software Maintenance (ICSM'05), 2005.

[22] Kaczor, Y. Guéhéneuc a S. Hamel: *Efficient Identification of Design Patterns with Bit-vector Algorithm*, Proceedings of the Conference on Software Maintenance and Reengineering (CSMR), 2006.

[23] C. Park, Y. Kang, C. Wu, a K. Yi: *A static reference analysis to understand design pattern behavior.*, In Proceedings of the 11^{th} Working Conference on Reverse Engineering (WCRE'04),2004.

[24] Jing Dong, Yongtao Sun, Yajing Zhao: *Design Pattern Detection By Template Matching*, Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC), pages 765-769, Ceará, Brazil, March 2008.

[25] Syed Fahad Gilani, Michael J Gillespie, Andy Olsen, Benny Mathew, James Hart: *Visual Basic.NET Reflection Handbook*, Wrox 2002

[26] Jing Dong, Yajing Zhao, Tu Peng: *A review of Design Pattern Mining Techniques*, IJSEKE 2009

A Dodatek

Na přiloženém CD jsou k dispozici zdrojové soubory k aplikaci a k dokumentaci a také je zde adresář `DATA`, ve kterém jsou testované data. Nejsou zde však všechny testované data a to z licenčních důvodů.